2015

CASPaR: Congestion Avoidance Shortest Path Routing for Delay Tolerant Networks

Michael F. Stewart
Louisiana State University and Agricultural and Mechanical College

CASPAR: CONGESTION AVOIDANCE SHORTEST PATH ROUTING FOR DELAY TOLERANT NETWORKS

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science in System Science

in

The Department of Computer Science

by
Michael F. Stewart
B.S. in Physics, Louisiana State University, 1999
December, 2015

Acknowledgments

First, I would like to thank Dr. Rajgopal Kannan, my advisor. I am grateful for his guidance his patience and am proud to have been his graduate student. I would also like to thank Dr. Amit Dvir who helped me every step of the way throughout my research which truly would not have been possible without his help. Additionally, I would like to thank committee members Dr. Costas Busch for his guidance and unwavering support and Dr. Supratik Mukhopadhyay for his support and advice.

I would also like to thank Maggie Edwards who worked tirelessly to make sure that all ducks were neatly organized and for her advice that was offered when ever I asked. I wish to extend my deepest gratitude to Dr. Bijaya Karki for supporting my research. In addition, I would like to thank to Dean Massé, Nick Davis and the rest of the Louisiana State University Graduate School Staff for their help and support.

None of this would have happened if it were not for Dr. T. Gregory Guzik for encouraging me to go back to school and to Dr. John Wefel and Dr. Mike Cherry for their complete support. I am extremely grateful. I would also like to thank my friends and co-workers, Douglas Granger, Bethany Broekhoven, Amir Javaid, Nick Cannady, Colleen Fava and Craig Jones who have all offered their support and have talked theory with me whether they wanted to or not.

To my family, Aimeé, Jacob, Sean and Elle, I am forever grateful for your patience, encouragement and love throughout this process. I love you all dearly and would not have endured if it weren't for your support. You make everyday wonderful and you brighten my world.

Table of Contents

List of Tables

List of Figures

Abstract

Unlike traditional TCP/IP-based networks, Delay and Disruption Tolerant Networks (DTNs) may experience connectivity disruptions and guarantee no end-to-end connectivity between source and destination. As the popularity of DTNs continues to rise, so does the need for a robust and low latency routing protocol capable of connecting not only DTNs but also densely populated, dynamic hybrid DTN-MANET. Here we describe a novel DTN routing algorithm referred to as Congestion Avoidance Shortest Path Routing (CASPaR), which seeks to maximize packet delivery probability while minimizing latency. CASPaR attempts this without any direct knowledge of node connectivity outside of its own neighborhood. Our simulation results show that CASPaR outperforms well-known protocols in terms of packet delivery probability and latency while limiting network overhead.

Chapter 1
Introduction

1.1 DTN Background

A Delay Tolerant Network (DTN) is defined to be a network where communication between nodes is not guaranteed and a route is not always available for a packet to travel from source to destination. Communication between nodes may go down for any number of reasons. It may be due to node mobility and broadcast range or possibly due to the environment in which the devices are deployed. For example, consider a floating sensor network designed to measure wave height scattered in some area of the Pacific. The sensors are capable of communicating amongst themselves but in order to upload data to servers for permanent storage, they must get their data messages to one of only a handful of satellite transceiver relays scattered throughout the network. In order to do so, they must route data packets over any number of sensor relays. Unfortunately, the inherent nature of the waves makes it almost impossible for static routes to exist since relay nodes in wave troughs can not communicate with each other. New routes constantly have to be developed in order for packets to reach the satellite transceiver nodes. To further complicate communication, satellite-transceiver relay-node movement is random, in this case controlled by ocean waves and currents. The relays may end up congregated close together or scattered far apart from each other. Some may drift out-of-range. Regardless, the goal of the DTN portrayed in this example is to collect and log as much wave height data to the servers as possible, and do so for as long as possible, a typical requirement of DTNs.

In many cases DTNs are made up of low-power nodes and efficient use of energy is important to extend the life of the network. In addition, DTNs are often defined

by nodes that have limited storage capacity relative to nodes in a more traditional network. The wave-height experiment is a prime example of a network where each node must be expendable and therefore cheap to produce with a limited capacity for memory and power. For this reason, transmission power-consumption should be conserved either by limiting the number of broadcasts or limiting the range of broadcasts or both. This is a fundamental design requirement for DTN routing protocols.

DTN routing protocols must be able to deal with communication disruptions by holding onto packets and waiting for routes to be re-established, an attempt to facilitate communication where connected paths do not always exist (attempts to use conventional Mobile Ad-Hoc routing (MANET) protocols such as reactive [1], proactive [2], and hybrid [3] approaches have resulted in failure). This is because DTN protocols must adapt a "store and forward" approach, either as single or multi-copy routing protocol. It must do this under the constraints of low-power and small-memory. Also, total network capacity must be large enough and utilized efficiently enough to account for extensive message buildup in order to not drop packets or drop as few undeliverable (due to unconnected routes) packets as possible. This is inevitable considering nodes may be separated for long periods of time.

These constraints, limited connectivity, low power and small memory should not alter the overall goal of delivering messages to their destinations as quickly as the DTN will allow. Each time a message is delivered, it is removed from the network, power is conserved and room is made available in network buffers for new messages. If packets can be delivered quickly without consuming unnecessarily large amounts of network resources (power and memory), the network may be made to look more like a Mobile Ad-hoc Network (MANET) often and like a DTN

only when necessary. This is an important but challenging problem [4] especially considering that DTN devices are increasingly being integrated into our everyday lives.

1.2 DTN Routing Protocols

Liu et al. [5] defined an DTN organizational chart that divides DTN protocols into two major categories, forward-based and flood-based. A re-creation of that chart is shown in Figure 1.1. The forward-based strategy keeps a single copy of each message in the network. This type of DTN routing can be further broken down into 3 categories: infrastructure-based, prediction-based and social-based. An infrastructure-based approach is defined by the use of mobile agents that work to deliver messages across disconnections in the network. Social-based schemes rely on knowing the social behaviour of nodes in a network and to apply that knowledge in order to predict future movement while prediction-based routing uses historical knowledge to predict node movement.

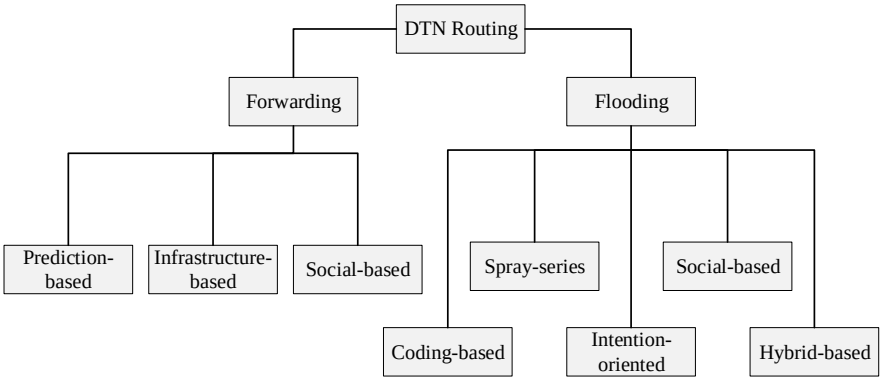


FIGURE 1.1. Classification of DTN Protocols: Shows a representation of a DTN classification system.

Flood-based strategies take the opposite approach and duplicate messages spreading them across the network. This strategy can be broken into several categories: spray-based, social-based, coding-based and intention-oriented. A separate hybrid approach is also described. The spray-based approach applies a two-phased algo-

rithm, a spray phase where some number of message copies are transmitted and a wait phase where nodes rely on a direct-delivery styled approach. The social-based flood approach is similar to the social-based forward approach but multiple copies are created to increase the likelihood that messages are delivered. The *BUBBLE* protocol [6] is a prime example of a social-based flood algorithm. The coding-based approach divides messages into smaller fragments, floods the network with them and then relay nodes recombine fragments and forward them. Once all fragments reach the destination, they are decoded and re-built into the original message.

Protocol	Packet Priority	Packet Copies	Forwarding Mechanism	Congestion Control
Direct Delivery	based on chance meeting between Src and Dst	none	must meet destination	none
Epidemic	based on difference between neighbors' Qs	unbounded	flood-based	none
Back pressure	FIFO or LIFO or WFQ	some, based on delivery predictability	delivery predictability flood-based	reactive buffer availability
PRoPHETv2	based on delivery predictability	some, based on delivery predictability	delivery prediction cost-based	none
MaxProp	n-hops, cost-based	some, TTL-base deletion	historical connectivity cost-based	none
Spray and Wait	FIFO or LIFO then those that have copies left	< 10	measured binary flood-based	none
Shortest Path	shortest path, oldest	none	Dijkstra-based semi-omnipotent	none
CASPaR	lowest cost, oldest	none	historical route and congestion cost-based	proactive, buffer availability
CASPaR-MP	lowest cost (multipath), oldest	none	historical route and congestion cost-based	proactive, buffer availability

FIGURE 1.2. DTN Protocol Table: Lists the primary attributes and differences between the DTN protocols testing during this thesis.

1.2.1 Direct Delivery

The most basic form of the forward-based (one-copy) DTN routing strategy is *Direct Delivery* [7]. Nodes forward messages to destinations only when they come in contact (within range) with the destination. This means that messages are shared directly between source and destination nodes. This method relies completely on chance meetings between source and destination nodes and can be quite useful since its delivery rate provides the probability that two nodes come in contact if all node movement is based on a random walk algorithm. This is a one-copy

algorithm and therefore makes efficient use of queue buffer space however, due to its overly simplistic routing scheme, it does not perform well unless nodes in the network encounter each other often.

1.2.2 Epidemic

The basic form of the flood-based DTN routing protocol is *Epidemic* dissemination [8], a multi-copy algorithm that can offer low delivery delay, but can be prohibitively expensive since it consumes a considerable amount of network resources due to excessive message duplication. *Epidemic* works in the following manner: when two nodes meet, they share buffer packet content information. Using the shared information, the nodes determine which packets they already have, those that they do not and those its neighbor does. The pair then exchange the necessary packets so that they both have the same packets in their buffers once the sharing transaction is complete. This process is repeated each time nodes come in contact with each other. While this approach has proven to work well under comparatively lower network loads, our results show that high network loads can render *Epidemic* routing completely ineffective.

1.2.3 PRoPHET

Lindgren et al. [9] presented a probabilistic flood-based routing protocol *(PRoPHET)*, the operation of which is similar to *Epidemic* except that information about "meetings" is used to update the internal delivery predictability vector used to decide which messages are delivered to other nodes. Each node calculates a delivery predictability and forwards messages only if the encountered node has higher delivery predictability than itself. Naziruddin and Pushpalatha [10], improved *PROoHET*'s efficiency in terms of buffer related constraints over the network.

1.2.4 MaxProp

Burgess et al. [11] proposed a history-based method [4] that relies on prioritizing packet transmission and drop scheduling, (*MaxProp*). Queued packets are divided into two groups; those packets that are below some n-hop threshold and those whose hop count is greater. New packets that haven't traveled far are given priority and as a result, newer packets are guaranteed a delivery opportunity. Those queued packets above threshold n are prioritized based on estimated cost to destination defined by the cost function:

$$c(i, i+1, ..., d) = \sum_{x=i}^{d-1} [1 - (f_{x+1}^{x})] \tag{1.1}$$

where c represents the cost across nodes $(i, i+1, ..., d)$ and f_{x+1}^{x} represent the cost or edge weight between neighboring nodes x and $x+1$.

However, this algorithm requires a large buffer and energy consumption, and suffers from severe contention. Also, a potential latency problem might arise from the preferential treatment given to low-hop-count packets. If the destination of a new packet is unreachable, wasted effort may needlessly be applied to packets with low hop counts.

1.2.5 Spray and Wait

The multi-copy, flood-based *Spray and Wait* protocol presented by Spyropoulos et al. [12] has been shown to outperform all existing schemes with respect to both average message delivery delay and number of transmissions per message delivered. However, it requires a large buffer. *Spray and Wait* has 2 phases of operation; a spray phase and a wait phase. *Spray and Wait* has a couple of different variants based on the number of packet copies disseminated. We've run *Spray and Wait* in binary mode which is described here. In binary mode, source node create L copies of each message. Nodes, upon meeting a node with no copies of a specific message

deliver, $\lfloor n/2$ copies to its neighbor. This is repeated until it is left with a single copy at which time it switches to the wait phase of delivery where it behaves like the *Direct Delivery* protocol.

For the simulations performed here, the initial number of copies was set to be 6. This means that if node n created a message, it would make 6 copies of the message. It would then deliver 3 of those to node l_1^1, the first node it meets, 1 to l_1^2 the second node it meets and one copy to l_1^3. At this point, the source is left with a single copy and reverts to wait mode and behaves as a *Direct Delivery*-type protocol. Now node l_1^1 would deliver 1 copy to node l_2^1 and another to l_2^2 leaving all 6 nodes: n, l_1^1, l_1^2, l_1^3, l_2^1, and l_2^2 with single copies and operating in direct-delivery mode.

It is clear why this algorithm can be successful especially in a small-map, random walk type simulation as in our results. It increases the probability of contact significantly. By distributing 6 copies to 6 different nodes, the packet has 6 times the likelihood of intersecting the destination node. If there is an option to stop delivery on those packets that have already been delivered, then it can also increase its efficiency. However, in a non-random node movement scheme, the binary mode *Spray and Wait* scheme might not perform so well. This is because at worst case, *Spray and Wait* is simply *Direct Delivery* with a higher probability of node-destination encounters but it can not break out of that mold. Also, it will be come less effective due to buffer overflow and ultimately dropped packets at higher network loads.

1.2.6 Backpressure and LaB

Backpressure routing [13] forwards packets along links with high queue differentials. Dvir and Vasilakos [14] presented a backpressure-based routing protocol for DTNs with link weights. Ryu et al. [15] considered nodes clustered in groups and used mobile relay nodes to ferry messages across groups. The authors [15] proposed

a two-level routing scheme, one intra via backpressure routing and one intre via source routing. However, backpressure algorithms do not take into account shortest routes. Alresaini et al. [16] aim to avoid backpressure's long delay in cases of low traffic by using a hybrid approach such as the social based forwarding algorithm presented in [6].

1.3 DTN Congestion Control

Congestion is caused by overuse of bandwidth within the network. Depending on the topology of the network, congestion can be a localized phenomenon or wide-spread. If congestion is localized then possibly the most effective means of bypassing it is to go around (avoid) it. If it is more widespread then there is no choice but to wait for it to subside; packet priorities being equal. Because DTNs do not behave as a continually-connected network, a typical approach to congestion control, Transmission Control Protocol (TCP) for example, does not work [17]. Congestion control must be designed into the routing protocol and avoided. The authors of [17] put forth a DTN congestion control taxonomy to classify congestion control techniques. Figure 1.3 shows a re-creation of the diagram that describes the taxonomy.

The taxonomy is divided into 8 main groups. The first, congestion detection, can be segregated into 3 categories: network congestion where the nodes try to detect congestion based on current throughput versus maximum throughput, buffer availability where nodes attempt to detect network congestion based on available space in neighboring buffers and drop rate where nodes base congestion on packet drops. Another group is the control type and is partitioned into 2 categories: proactive congestion control which aims to prevent congestion from occurring and reactive congestion control which to reacts to reduce congestion once detected. The routing group indicates whether the congestion control mechanism is routing

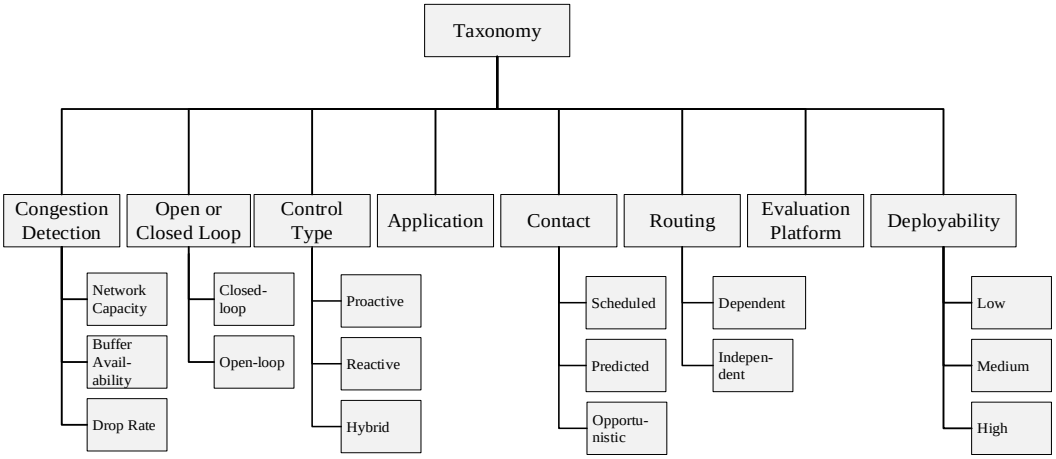


FIGURE 1.3. DTN Congestion Control Taxonomy: As first proposed in [17], this figure shows the proposed DTN congestion taxonomy which we use to help classify *CASPaR*'s congestion control mechanism.

protocol dependent or independent and the contact group describes how contact between nodes in the network come in contact: in a scheduled fashion, a predictable fashion or completely randomly (opportunistically). The last group, deployability, describes how realistically deployable a congestion mechanism is.

1.4 Applications

The list of potential applications for a high-bandwidth capable, efficient and reliable DTN routing protocol continues to grow and the networking boundaries between them is blurring. Some DTN applications where CASPaR would be effective are described here.

1.4.1 Vehicular Network

Consider a vehicular network [18] that allows vehicles, traffic sensors, traffic control centers, gas stations, restaurants and all else travel, traffic and automobile related to communicate with each other on one network. How might these vastly different entities communicate? The travel stops such as gas stations, restaurants and hotels are all capable of TCP/IP base communication. But, the automobiles and traffic sensors form a network in which end-to-end connectivity isn't guaranteed, a DTN. In the not so distant future, autonomous vehicles will have to communicate with

each other and with traffic sensors to efficiently and safely navigate. Cars might also communicate with gas and service stations and negotiate fueling or servicing options or appointments. Traffic sensors will route cars to less congested roadways, time lights to increase traffic throughput and ultimately ease traffic congestion and make traveling more safe. The cost effective nature of wireless communication makes it an obvious choice for vehicular networks. An elegant, efficient and simple networking solution is to create a mesh network from the sensors and vehicles themselves so that each and every one is responsible for relaying packets.

1.4.2 Interplanetary Network

Whether it be human or robotic, we are launching more things into space now than ever before. One commonality is that each of these spacecraft will have to communicate with home. As we travel further from Earth, a far-reaching space or interplanetary communication network (a deep-space network, DSN) will be needed (and already exists in some form [19]) to facilitate and route packets between home and these spacecraft potentially separated by millions of miles.

An example of the populating of our near-Earth space environment is the coming CubeSat revolution. David Pierce, senior program executive for suborbital research at NASA states that, "CubeSats are part of a growing technology that's transforming space exploration" [20]. A CubeSat is a small satellite approximately 10 centimeters cubed for a 1U (unit) sized model. They can be built in 2U, 3U, or 6U sizes as well. They weigh approximately 3 pounds per unit. Many are typically launched at once usually as axillary payloads making them launch-cost-effective. The number of small, sometimes tiny, space satellites are on the rise. These devices often can be designed and built for far less than their large heavy counterparts. Maybe more importantly, they can be launched for fractions of the cost.

Because technology has allowed for the miniaturization and greater efficiency of integrated circuits that perform all types of tasks, it is now feasible to build very small, relatively low-power communication devices that can be spread across vast regions of space building an interplanetary network to complement the existing DSN. This network is absolutely necessary if communication in and around our solar system is to be realized. Ultimately, localized interplanetary space travel hinges upon reliable communication.

1.4.3 Mesh Networking Solutions

General mesh networking solutions are being in explored in novel ways. An idea now coming to fruition is Google's Project Loon [21]. Loon, aims to provide those in developing regions of the world internet access by flying LTE payloads (mini-cell towers) right above the tropopause at an altitude of 20 kilometers. Many of these balloons will be launched over a large area and provide mesh network coverage for anyone with an LTE enabled device in that region. Google Loon balloons currently achieve 100 days at float and have communication ranges on the order of 400 kilometers when several balloons are meshed in a single network. Float time and broadcast range are improving and as they do, LTE communication cost drops; possibly to the point that it will be cost effective to deploy communication balloon mesh networks in developed areas of the world; maybe here in the U.S. in congested areas where cell towers aren't cost-effective or even feasible to construct.

This is a specific example of a broader push towards wireless, mobile, mesh communication allowing complete high-bandwidth connectivity between devices that may or may not always be connected. Routing techniques must be able to accommodate the potential for parts of the network to be delay and disruption tolerant and to bridge the highly mobile, mobile and non-mobile portions.

1.5 Motivation

Currently, there is not a *one size fits all approach* to MANET and DTN networking in general but maybe it is time to start working on it. Wireless communication and sensor devices perform all sorts of jobs but as wireless and sensor technology decrease in cost, their numbers will increase causing individual DTN islands to grow in numbers. As these islands begin to overlap they will merge. This process will repeat and as it does the DTN footprint will grow and its bandwidth requirements will expand. The clear delineation between MANETs, DTNs and even the internet will blur as devices once considered separate join the global network (globnet). This growth may continue until it reaches off-planet and the inter-planetary network one day joins the globnet. It is clear that there is a need for a common, efficient, robust routing protocol that can link these networks and account for specific DTN characteristics such as contact information, mobility pattern and network resources (storage space, transmission rate, and battery life).

1.6 Research Goals and Requirements

The development of a multi-purpose, one-copy DTN protocol that addresses congestion avoidance, shortest path routing and is capable of operating efficiently in a high-load network is the motivation behind the Congestion Avoidance Shortest Path Routing protocol (*CASPaR*). The algorithm is defined by the following developmental guidelines:

1. Do not duplicate packets.
2. Route deliverable packets, move undeliverable packets 'closer' to their destinations and hold onto packets when prudent to do so.
3. Integrate congestion avoidance and bottleneck minimization into the design.

The goals of *CASPaR* are:

1. Learn direct routes to destinations when possible.

2. Avoid congestion.

3. Dynamically correct routes as the network topology changes.

4. Minimize latency and maximize delivery by moving packets over those newly discovered routes.

To do this, *CASPaR* must negotiate node queue differentials between neighbors similar to back-pressure algorithms and map shortest paths without explicitly discovering them. Here we present preliminary results showing that *CASPaR* accomplishes these goals.

1.6.1 Derived Requirements

Figure 1.4 shows a flow diagram that presents the overall goal and constraints of *CASPaR*. It is a tool used to help derive the requirements of *CASPaR* whose overall goal is to deliver all packets as quickly as possible, for as long as possible and as *cheaply* as possible under the restrictions of low memory, low power and periodic disconnected nodes; typical restraints placed on a DTN.

Following the flow of the diagram shows, for example, that to deliver packets quickly, *CASPaR* must move them closer to or directly to their destinations at each update interval but avoid congested routes when doing so. To avoid congestion, packets must be distributed evenly across the network topology and to accomplish this, probable routes must be known and queues must be balanced. Probable routes must be known in order to move packets closer to their destinations as well and to accomplish both of these objectives, queue and route information must be shared between nodes.

Because DTNs are defined by their disconnected paths, packets must be stored in the network until a route becomes available. Therefore, packets must be dropped

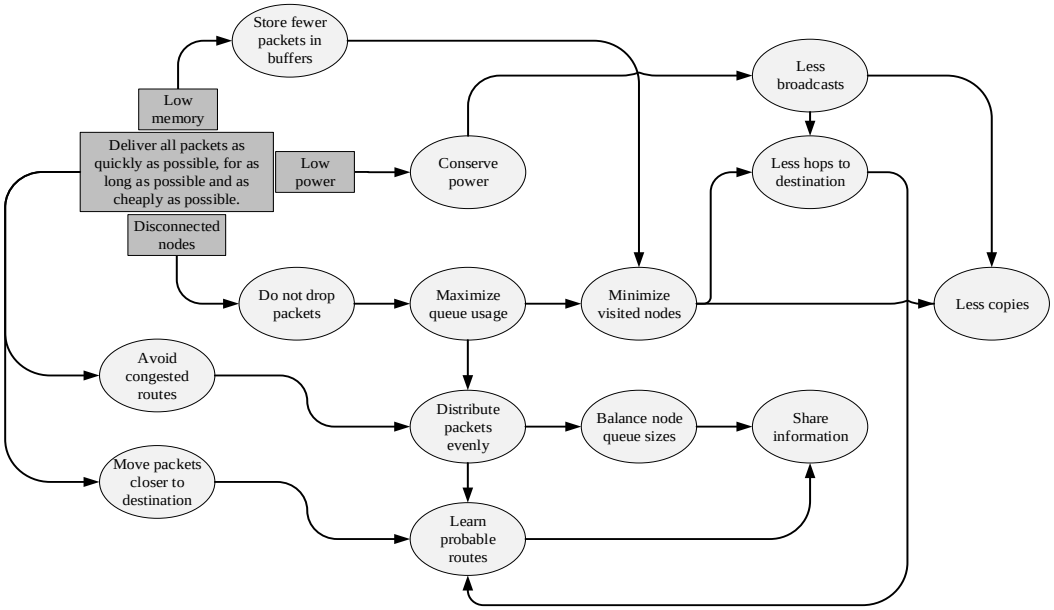


FIGURE 1.4. Goals and Requirements Chart: Lays out the overall goal and constraints of *CASPaR* and the objectives and requirements that are derived from that overall goal.

only as a absolute last resort which dictates efficient queue usage. Again, this requires an even distribution of packets across the network but also the minimization of node visits which means that fewer packet copies should be made and fewer visited nodes enroute to a destination. This also requires that probable routes be discovered.

DTNs often have low-power restrictions and therefore energy conservation is a must. Power consumption is driven by the number of transmissions so by minimizing broadcasts, power will be conserved. To do this, 2 things must be constrained: 1) the number of hops a packet must make to get to its destination and 2) the number of duplicate packet copies must be minimized. Both of these restraints require that probable routes to destination nodes are *learned* .

Lastly, DTNs often have low-memory restrictions which means that few packets can be stored in each node's queue and forces efficient queue management. This means that the number of nodes a packet traverses must be minimized which requires that there are fewer hops to the destination node and fewer copies of

packets. But to deliver packets quickly under these constraints, probable routes to destination nodes must be discovered.

This diagram identifies the requirements of an efficient DTN routing protocol design. An efficient protocol must share information between neighbors, makes few packet copies, if any, learn probable routes to destinations so that packets can be distributed over the network to avoid congestion and minimize routing hops.

1.7 Research Methodology Overview

The research methodology of the *CASPaR* study, whose goal is to develop a single-copy DTN routing protocol whose predicted routes avoid network congestion and form direct routes to destinations, is described in this section. First, existing DTN routing protocols were studied to better understand the DTN problem and to determine which of the existing protocols should be used to compare with *CASPaR* (the comparison protocols are referred to collectively as *comparison protocol*). Once the DTN problem and existing solutions were better understood, a preliminary version of the *CASPaR* algorithm was developed, tested (by simulation) and the results compared with *comparison protocol* results (using the same simulation parameters). As problems and glaring inefficiencies with the *CASPaR* algorithm were discovered, they were studied, fixed and the algorithm was modified, tested and the results compared again. This refinement process included discussions amongst committee members and a few times involved tweeking the basic analytical model that the *CASPaR* algorithm is based upon. The entire process of testing, comparing and refining was repeated over several iterations until the final version of the *CASPaR* algorithm was created.

Once a *final CASPaR* algorithm was developed, 'official' protocol testing began. Testing involved a very specific network simulation where the only changes made between runs were the routing protocol, the random number generator seed and

the size of node buffers. The key comparison results were chosen to be delivery probability ($messagesdelivered/messagessent$), average message latency (the average amount of time it takes all packets to be delivered), and hop count (the average number of nodes a message encounters before being delivered to its destination). Special analysis attention was paid to the latency results which required simulation data reports accounting for all transmitted messages including their full paths from source to destination, their start times and their latencies.

To more completely gauge the high-performance behaviour of *CASPaR*, an *ideal* single-copy routing protocol was required for comparison. A routing protocol that *knew* the single shortest path between all source and destination nodes for all queued messages at time t, one that could minimize hop count and latency while maximizing delivery probability was needed. The *Shortest Path* routing algorithm was developed to fill this need. To gauge low-performance behaviour, a *basic* single-copy routing protocol was required. The existing *Direct Delivery* routing protocol fit this requirement well. Remember that *Direct Delivery* is the routing protocol whose algorithm is one of self-delivery. That is, the only way a message is delivered is if the source and destination come within broadcast range or each other. Together, these two protocols provide the extreme case simulation comparison results. The other protocols that are simulated and compared in this study are:

1. *Epidemic (EPI)*: Due to its potentially high overhead [8].

2. *Prophet with Estimation (PRO)*: Due to its probabilistic routing [9].

3. *MaxProp (MP)*: Due to it documented high-performance [11].

4. *Backpressure LaB (LaB)*: To compare with *CASPaR*'s backpressure-like mechanism [13] [14] [15].

5. *Spray and Wait (SaW)*: Due to its high performance [12].

1.8 Thesis Outline

The rest of this thesis is organized as follows:

1. Chapter II describes the *CASPaR* algorithm by detailing the analytical model it is based upon, providing the algorithm, describes an important variant and provides an example.

2. Chapter III describes the simulation, the simulation methodology, the specific simulator used including its input, execution style and report mechanisms.

3. Chapter IV provides the simulation results for delivery probability, latency, overhead, and hop count. Some results and explanation of network load balancing is also provided as well as a more detailed investigation into message or packet latency.

4. Chapter V has some concluding remarks and discusses some ideas into possible future study regarding *CASPaR*.

Chapter 2
CASPaR

CASPaR is a one-copy routing protocol that attempts to route packets over the shortest, least congested paths. *CASPaR* consists of two interdependent mechanisms: 1) direct routing and 2) congestion avoidance. The algorithm is designed to route packets over connected paths and employs a routing-protocol-dependent, proactive congestion-avoidance mechanism [17] that uses an open loop congestion control scheme based on buffer availability and historical connectivity knowledge. This allows for alternate route discovery avoiding congestion buildup. Ultimately, congestion avoidance takes precedence over routing forcing a direct-delivery-like mode of operation during heavy traffic. Except for their 1-hop neighbors, nodes have no knowledge of other nodes in the network.

2.1 Principle of Operation

All nodes maintain an estimated cost ($C_n^c(t)$) to deliver packets to each destination node c. This cost attempts to track the least congested and shortest paths to each destination c based on historical knowledge of connectivity to the destination and the waiting times of packets to c in node n's queue. The process by which $C_n^c(t)$ is calculated begins with the broadcast of a Request For Costs (RFC). All nodes participate in the RFC transaction process when one of three things occurs: 1) a packet has just been received from a neighbor, 2) a packet has just been created or 3) the RFC periodic timer expires. Neighboring nodes, upon receiving an RFC, respond with their destination cost table which contains a list of all destinations and the cost to send a packet to that destination. If node n's estimate of delivery costs to c is the lowest amongst its neighbors, then n holds onto these packets

in its buffer until it either meets a neighbor with a better (lower) estimate or is connected to c (we use a preference factor of 0.9 to give a slight preference to node n holding onto these packets). Priority transmission is given to those packets whose destination are neighboring nodes. The effect of periodic updates is a more accurate network congestion and connectivity model and since routes depend upon a neighbor's total transmission costs, frequent updates produce a more applicable model (similar to distance vector routing in wired networks [1]). Nodes have no direct knowledge of the state of the network outside of its own neighborhood. But due to the propagation of costs, each node gains an approximate network-wide perspective allowing for effective packet routing.

2.2 Model

Path congestion and route connectivity are modeled by minimizing the delivery costs along some multi-hop path from source to destination and is characterized by two convoluted parameters: The first is *Proximity Measure*:

$$\Theta_n^c(t) = \frac{Q_n^c(t)}{T_n^c(t)} \tag{2.1}$$

$\Theta_n^c(t)$ is a value between 0 and 1 where 1 indicates nodes n and c are connected and 0 means they were never connected. $T_{n,t}^c$ is incremented at every time step and $Q_{n,t}^c$ is set equal to $T_{n,t}^c$ as long as nodes c and n remain connected forcing $\Theta_n^c(t)$ equal to 1. Once disconnected, $\Theta_n^c(t)$ begins decreasing linearly in time. Periodically both Q and T are reset to some initial values that represent a default measure of connectivity. The second parameter is the *Net Destination Queue Waiting Time*:

$$W_n^c(t) = \sum_{i=0}^{N} (\mathcal{T} - a_{n,i}^c) \tag{2.2}$$

where $\mathcal{T}$ is the current time and $a_{n,i}^c$ is the arrival time of packet i at node n destined for node c. The queue waiting times of packets are used as a proxy for

congestion as opposed to backpressure which uses queue size differentials. Hence, we model delivery costs as an exponentially increasing function of net waiting times of packets with an increasing discount factor based on connectivity probability. The estimated delivery costs to c via n are calculated as:

$$C_n^c(t) = W_n^c(t)\dot{(}1 - \Theta_n^c(t)) + C_n^c(t-1) \tag{2.3}$$

CASPaR's, estimated delivery cost is calculated while explicitly setting transmission costs between 1-hop neighbors to 0. This emphasizes routing along a connected path between source and destination *when one exists*, and routing to balance congestion in the network *when connected paths do not exist*. Setting 1-hop transmission costs to 0 has the following effects: 1) If a connection from source n to destination c exists then the delivery cost will be 0 everywhere along that path regardless of path length sinking packets directly to destination c (see line 24 from Alg. 1) and 2) If a connection from source n to destination c does not exist then packets to c will be spread over the network based on congestion, radiating outwards towards the destination. Eventually when one of these nodes becomes connected, a direct path to c is created and packets quickly flow down-gradient to their destinations.

In addition to $\Theta_n^c(t)$ being set to 1, the historical cost, $C_n^c(t-1)$, is reset to 0 when nodes n and c become connected at time t. From the definition of W earlier, the marginal increase in net waiting times at each time step are a function of queue size to c. Thus as can be seen from the expression above, lightly congested nodes along short paths to the destination are favored (the more recently a node is in contact with the destination and the smaller its queue size, the lower the transmission cost) and therefore the net effect of the algorithm is to reinforce delivery on short, less congested paths.

Proximity Measure and *Net Destination Queue Waiting Time*, parameterize not only the shortest but least congested paths. The *Proximity Measure* attempts to minimize the path length from source to destination while *Net Destination Queue Waiting Time* pulls packets towards neighboring nodes with the smallest queues (similar to a backpressure mechanism [16]) minimizing routing across congested paths. This technique develops routes that *chase* the destination, ultimately catching and creating short paths from source to destination.

Packets are transmitted in a lowest-cost first, longest-queue-waiting-time second, priority order. More simply put, the oldest, cheapest packets are transmitted first. Also, a minimum node loop counter to force a Minimum Loop Size (MLS) is integrated into the *CASPaR* algorithm to avoid packets repeatedly traversing the same nodes. The MLS is defined to be the minimum number of consecutive unique nodes that must exist in a routing path before a packet is allowed to revisit a node. The MLS is set to 5 for all simulations presented here.

2.3 Algorithm

Request for Costs is executed both periodically and upon the receipt or creation of a packet. The range status, measure of proximity, net destination queue waiting time and total transmission costs are recalculated upon each call (see Alg. 1 and Table 2.1).

2.4 Multi-path Variant

Several variations of the *CASPaR* protocol were designed and tested during this DTN study. Two that emerged as notable candidates are the *CASPaR* and *CASPaR-MP*. The 'standard' variant, defined by single path costing is designed based on costs to route packets to the neighbor that replies with the lowest relay cost based on single routes to destinations. It is referred to as single-path costing or the

TABLE 2.1. Algorithm Definitions

$C_n^c(t)$	The transmission cost for all packets destined for node c that reside in node n's queue at time t.
$W_n^c(t)$	The net destination queue waiting time is the amount of time that all packets destined for node c have been resident in node n's queue at time t.
$\Theta_n^c(t)$	Proximity Measure that is analogous to the elapsed time since nodes n and c were within k-hop radius of each other such that $0 < \Theta_n^c(t) \leq 1$. When nodes n and c are connected, $\Theta_n^c(t) = 1$. If nodes n and j have never been connected, $\Theta_{n,j}(t)$ is 0.
$R_{n,t}^c$	The range status between node n and destination c at time t. If node c resides in the k-hop neighborhood of node n at time t, the range status, $(r_{n,t}^c)$, is set to true. Otherwise it is set to false.
$a_{n,i}^c$	The arrival time of the ith packet at node n destined for node c.
$T_{n,t}^c$	A tick counter which is incremented upon each bid period. It is reset to some default measure of connectivity periodically.
$Q_{n,t}^c$	Counter incremented each time nodes n and c are not neighbors. It is reset to some default measure of connectivity periodically.
τ	The current time.

CASPaR-SP variant. Since it is the standard algorithm it is always referred to as simply *CASPaR*.

A slight modification of *CASPaR* takes steps to distribute packets more widely over the network as an enhanced congestion avoidance technique and is referred to as the multi-path or simply *CASPaR-MP*. Instead of calculating costs based on a single route from a relay node to a destination, *CASPaR-MP* takes into account all possible routes to a destination during the cost determination process.

The multi-path designation may be somewhat of a misnomer. It does not mean that messages are split and sent across different routes towards their destination nor does it mean that a relay node will alternate between routes when sending messages to some set destination. It means only that route costs are calculated

Algorithm 1 The CASPaR Algorithm

function UPDATE RANGE STATUS
  for all destinations **do**
    if destination c is within 1-hop of node n **then**
      $R^c_{n,t}$ = true
    else
      $R^c_{n,t}$ = false
function UPDATE MEASURE OF PROXIMITY
  for all destinations **do**
    $T^c_n(t) = T^c_n(t) + 1$
    if $R^c_{n,t}$ **then** $Q^c_n(t) = T^c_n(t)$ ▷ Periodically reset to default values
    $\Theta^c_n(t) = \frac{Q^c_n(t)}{T^c_n(t)}$
function UPDATE QUEUE WAITING TIME
  for all destinations **do**
    $W^c_n(t) = \sum_{i=0}^{N}(\mathcal{T} - a^c_{n,i})$
function UPDATE DELIVERY COST
  for all destinations **do**
    if $R^c_{n,t}$ **then** $C^c_n(t-1) = 0$
    $C^c_n(t) = W^c_n(t)(1 - \Theta^c_n(t)) + C^c_n(t-1)$
function REQUEST FOR COSTS
  Update Range Status ()
  Update Measure of Proximity ()
  Update Queue Waiting Time ()
  Update Delivery Cost ()
  $C^c_n(t) = \frac{9}{10}\dot{C}^c_n(t)$ ▷ Calculate Self-Delivery Estimate
  for all nodes j in 1-hop range of node n, for all destinations c **do**
    Select $C^c_m = \min(C^c_j(t), C^c_n(t))$ and relay r accordingly
    Update $C^c_n(t) = C^c_m$ and either relay packet to node j or do not transmit if $r = n$

based upon all possible routes to each destination from some relay node instead of basing it on the single lowest cost route. It will, however, behave in such a manner as to allow for separate back-to-back messages to very likely be transmitted over different routes. This is demonstrated in Figure 2.1.

Take a network that consists of nodes n, j_1, j_2, and c and paths x, y and z for example. Node n wishes to deliver a packet to node c and must select the node that reports the minimum cost to be the relay. Node j_1 is connected to node c through two paths, x and y. Node j_2 is connected to node c through only one path, z. Out of all paths, x, y and z, z is the least congested and therefore the single *cheapest* route. However, because node j_1 can offer 2 *perceived* independent paths, node j_1's presented cost may be less than node j_2's depending on the cost-combination of the individual bids. In the case shown in Figure 2.1, x has a cost of 3, y has a cost of 3 and z has a cost of 2. The cost presented to n by j_1 to transmit a packet to destination c would be calculated in the following manner:

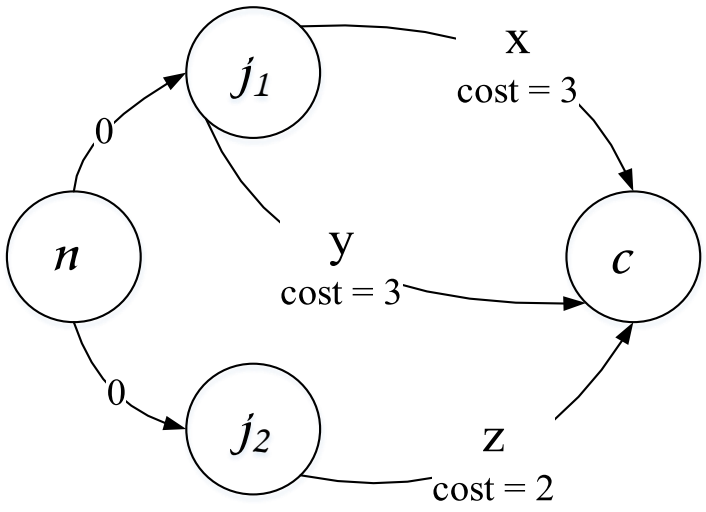
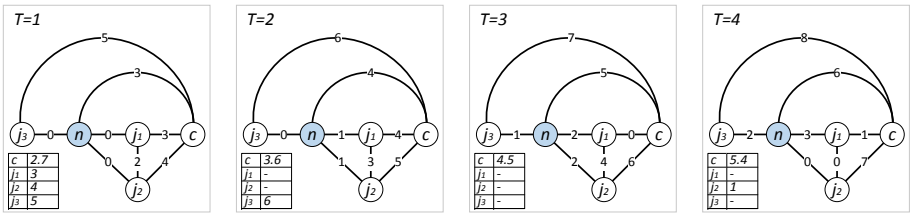


FIGURE 2.1. Multi-path Diagram: Shows the functionality of *CASPaR-MP*. Node j_1 has 2 routes, x and y to destination node c both at cost 3. Node j_2 has only one route z at a cost of 2. In single-path routing, node n would choose j_2 to send packets through since it has the single lowest cost route. However, in multi-path routing, node n could choose j_1 depending on the combined cost of its two parallel routes.

$$C_n^c(t) = 1/\sum_{i=0}^{N}(1/p_i) \tag{2.4}$$

where p_i is the i^th path.

In the scenario presented here, the cost reported to node n by j_1 is 3/2 which is less then the cost presented by j_2 which is 2 and therefore packets destined for c would be transmitted through relay j_1 at time t. Assuming this scenario, node n would choose j_1 to send packet-1 onto destination c. Lets assume that j_1 sends packet-1 through path x. Now lets say another packet, packet-2 is sent by node n to node j_1. Node j_1 re-processes a RFC and it is now likely, since packet-1 may still reside in the buffer of the node only 1-hop away from node j_1 along path x, that path y will produce the lower cost and hence packet-2 will be sent through path y towards its destination c. From this example, it can be seen how *CASPaR-MP* can easily be conformed into a routing protocol capable of splitting packets and send them across varying routes.



(a) Time T=1 (b) Time T=2 (c) Time T=3 (d) Time T=4

FIGURE 2.2. CASPaR Example Diagram: A *CASPaR* example iterating through 4 time periods and the transactions between a group of nodes in a small network.

It is shown in the results that *CASPaR-MP* does provide a slight advantage over *CASPaR*. However, it was not chosen as the standard because of its analytical complexity and minimal performance gains when compared to *CASPaR*.

2.5 Example

In the scenario presented in Figure 2.2, the weighted graph represents a small network. The vertices represent nodes: n, j_1, j_2, j_3, and c. The weighted edges represent the transmission cost between nodes (a weighted-edge of 0 represents neighboring nodes). In this scenario, node n is to deliver a packet to node c. Each panel represents a time-step and depicts a single RFC transaction. There are 4 panels starting at $T = 1$ and ending with the delivery of the packet at $T = 4$. Queue sizes aren't explicitly considered in the transmission cost and the measure of proximity is calculated using integers for simplicity. The self-delivery costs are multiplied by 9/10 as the algorithm is defined. At the bottom left-hand corner of each panel is the *destination cost table* showing all potential destination nodes and their associated costs.

At $T = 1$, node n broadcasts a RFC. Nodes j_1, j_2 and j_3 respond with their *destination cost tables* and node n compares them against its self delivery cost. These values are shown in the representative destination cost table: $c = 2.7$, $j_1 = 3$, $j_2 = 4$ and $j_3 = 5$. Since self delivery cost is the minimum, node n holds onto the packet even though it is unable to deliver it to its destination at this time.

Node n is unaware of the state of the network beyond its own neighborhood. After the RFC responses are received, node n learns its neighbors: j_1, j_2 and j_3 and that they can deliver a packet to node c for 3, 4 and 5 respectively. Node n deduces that a direct route to node c doesn't exist since a minimum delivery cost of 0 wasn't received. Node n selects itself as the relay since its delivery cost is the minimum.

At $T = 2$, node n again broadcasts a RFC but this time only node j_3 responds. The other nodes have moved out of range. Since the self delivery cost, incremented to 4 from time period $T = 1$ to $T = 2$, is still the minimum, node n again holds onto the packet.

Node n has no neighbors at $T = 3$ and by default its cost is the minimum and node n continues to hold the packet. Notice, that nodes j_1 and c become neighbors at this time. Unfortunately node n can not know this since it isn't connected to node j_1.

Finally, at $T = 4$, node n is a neighbor of node j_2 who responds with the minimum delivery cost of 1. Node n transmits the packet to node j_2 who will transmit the packet to node j_1; provided that nodes j_1 and j_2 are still neighbors once j_2 is ready to re-transmit.

The network is dynamic and can change quickly. The receipt of the packet from node n causes node j_2 to initiate its own RFC transaction that might reveal a route change. Other nodes potentially in the path might update their *destination cost tables* due to received and generated packets or because the update timer expired. A transmission cost of 0 reveals an end-to-end connection from current source to destination and can trigger an avalanche of packet transmissions towards the destination.

Chapter 3
Simulation

3.1 Purpose and Methodology

The purpose of the simulations was to compare the performance of *CASPaR*, its multi-path variant and 7 additional routing protocols as a function of buffer size. A realistic yet simple simulation was required that placed all protocols on equal footing. A relatively high data throughput was desired to stress the nodes in the network but the transmission rate was set to resemble typical LTE transfer rates of about 5 Mbps as measured from an actual LTE phone on AT&T's network in Baton Rouge, Louisiana. Systematic effects of the simulation were considered to ensure that A) there were no special case simulation runs for any of the tested protocols and B) if large variations existed in simulation results for a particular routing protocol and simulation scenario, they were known and their deviations accounted for. Therefore, the simulation had to be run multiple times but with different random number generator seeds to generate different results.

This called for a simple simulation scenario with few modifiable parameters as this thesis is an introductory study of *CASPaR*. The only parameters that were modified were the routing protocol, the buffer size and the random number generator seed for the node movement engine. The results had to include delivery performance in terms of probability, latency and overhead. The results also had to include all routes taken for all messages delivered including latencies and number of hops. These results were needed to perform more in-depth analysis on latencies as a function of routing.

Two candidate network simulators, NS-2 and ONE, were reviewed. The ONE simulator was chosen for its Java programming interface, its realtime simulation

GUI, because it is specifically geared towards DTNs and because it already has many of the standard DTN routing protocols contained within the installation.

3.2 The ONE Simulator

The Opportunistic Network Environment simulator (ONE) version 1.4.1 [22] was used for all simulations performed during this study. ONE is a graphical network simulator specifically designed for simulating DTNs. It comes with standard routing algorithms including *Direct Delivery*, *Epidemic*, *PRoPHET*, *Spray And Wait* and *MaxProp* all of which were simulated along with *CASPaR*. ONE provides a java programming interface complete with all classes required to design, develop, incorporate and simulate the behavior and performance of new routing algorithms. It is also capable of collecting and reporting network summary data that can be easily collated and analyzed.

With it, nodes can be created, placed within a blank or elaborate map and translated according to many different movement models. Some of the common ones are: random waypoint, map-based random waypoint and map-based routed movement models. There are also movement models designed specifically for different vehicles like cars and buses, and for different times of the day like work day hours and evening trends. While movement is being orchestrated, broadcast communication is simulated between nodes within range of each other. Each node has its own broadcasting time-slice and the selection rotation is randomized.

All simulations were run using the java runtime engine (*jre*) version 1.8.0_40 and all coding was written, compiled and run using the Eclipse Standard Software Development Environment *Version: Kepler Service Release* 1 *Build id:* 20130919 − 0819.

3.2.1 Input

Various input parameters are loaded at execution time in the form of a *parameter initialization file*. The *parameter initialization file* used for *CASPaR* simulations is provided in the appendix. These parameters define key simulation attributes. These key attributes include definitions for: overall scenario settings, broadcast settings, nodes and node movement, routing, event and message generation and summary reporting. The overall scenario settings include: the overall map size, the update interval, the number of different node groups, the random number generator seed to use for the movement model and the length of time that the simulation is to be run. The broadcast settings include: the radio interface used by a group of nodes, and the transmission range and speed of that radio interface. The node settings include: the number of nodes in a group, the movement model used by that group of nodes as well as the group movement speed and wait time. The routing parameters include: the type of router used by a group of nodes, its buffer size, and type and its message or packet time-to-live (TTL). The event and message generation settings include: the different event generation groups, the message generation rate for each group, the message size and the range of message source and destination addresses. The summary reporting parameters detail which reports are generated and various specific settings that a report might require. For example, the message location report which tracks the location of all messages requires the reporting granularity parameter set in seconds. This indicates the interval at which the location of messages is recorded.

The ONE simulator allows for various groups of nodes, radio interface and movement models to be defined as well as various groups of event generators. These groups can be mixed and matched to create a very versatile simulation. Many parameters can also be defined as a range. Movement speed, for example, can be

defined to be between 0.5 and 1.5 meters per second. When these degrees of freedom are combined, it allows for quite complex simulations. The default scenario, the Helsinki model for example, consists of various groups of nodes, some pedestrians, cars, and trams all traveling at appropriate speeds and moving according to map-based movement models where pedestrians walk on sidewalks, cars drive on roads and trams travel the same routes over and over on rail throughout the map of Helsinki. Pedestrians and cars have a different broadcast range and rate than do the trams. This scenario presents a more realistic scenario and much more complicated ones can be constructed.

The simulation environment can also be quite simple as well. The simulation model used for the *CASPaR* study is one such example. The movement model is a random way point where nodes randomly pick a point to move to then move to that point at some defined speed or range of speeds. Once they arrive, they wait there for some randomly determined amount of time then pick a new point and the process repeats until the simulation ends.

The *parameter initialization file* allows for multiple settings to be specified for most of the parameters by simply adding to a comma-delimited list bounded by brackets. See Table 3.1. When run in batch mode, ONE is capable of executing multiple iterations one after another. Upon each new simulation run, parameters in comma-delimited list format are iterated through one after the other and used as the input parameters for that run. If only a single setting is present for a parameter, the setting will be repeated for each execution.

3.2.2 Execution

Simulations can be run in either graphical or batch mode. A graphic mode simulation can be run from within the development environment and provides a graphical runtime view of the network grid and the movement of nodes within the grid. It

provides views to each node's queue and packet routing information. It is a good tool to learn the behavior of the routing algorithm being designed and tested. It does slow simulations down, consuming valuable CPU cycles, so for multiple simulations it is best to run in batch mode.

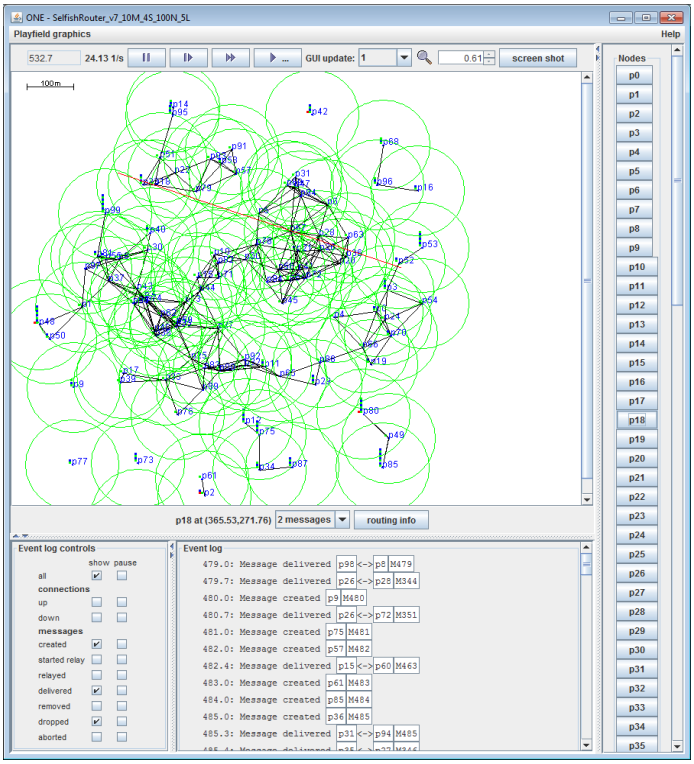


FIGURE 3.1. ONE Graphical Interface: The ONE graphical interface includes simulation control, a node movement view and packet routing information.

Figure 3.1 shows the ONE simulation GUI. Towards the top are the simulation control buttons: play, pause, speedup and step-through are all available. The network grid can also be resized. The upper-left shows the simulation elapsed time and the number of simulation cycles per second currently being executed. At the bottom left is the event log and event log control. The event log displays critical events such as message creation, delivery and drop as well as when connections are made or lost. The types of events that are displayed is controlled using the check boxes in the event log control. Along the right side of the GUI is the list

of nodes present in the simulation. The coordinate location, number of messages in queue and routing information for all messages to and from any node can be displayed by clicking on any one of the node numbers. This is an invaluable tool when designing, implementing and testing a new routing algorithm.

Batch mode allows multiple simulations to be run from a single command line execution by using the multiple simulation option -N where N is the number of simulation runs to be performed. As previously stated, batch mode also allows parameter settings to vary between runs but this requires proper construction of the *parameter initialization file*. Notice in Table 3.1 the *Group.bufferSize* parameter has 7 different buffer size settings comma-delimited within brackets. When run with $N = 7$, 7 *CASPaR* simulations would be run, each with a different buffer size; first 0.2 MB, then 0.5 MB, then 1.0 MB and so on through 30 MB. If N was set to 8, the settings just wrap around to the beginning of the list, and therefore the 0.2 MB buffer size setting would be used again. All multiple parameter settings function in this manner. For every run, the next setting in the list is used until the end of the list is reached at which point it starts at the beginning of the list. This allows building complex batch jobs to run many time-consuming simulations as opposed to running them individually.

3.2.3 Reporting

The ONE simulator offers many reporting tools that are engaged by simply adding the report name to the *parameter initialization file*. Detailed here are the couple reports used to produce the results discussed in this paper.

The Message Statistics Report, as shown in Table 3.2 contains a summary report of all nodes for an entire simulation. It contains the standard results used to produce the following plots as a function of queue buffer size: Delivery Probability, Overhead Ratio, Hop Count and Packet Latency. The key statistics used

TABLE 3.1. Example Parameter Initialization File: An excerpt from a *parameter initialization file* showing specifically how to vary parameters between multiple run batch mode execution. Anything behind a # is a comment.

Parameter = Setting(s) or Comment (behind #)
# Common group movement model
Group.movementModel = RandomWaypoint
# Set group router to be *CASPaR*
Group.router = [caspar]
# Set varying queue buffer sizes
Group.bufferSize = [0.2M; 0.5M; 1M; 3M; 5M; 10M; 30M]
# Output 6 standard ONE report
Report.nrofReports = 6
# Standard message statistics report
Report.report1 = MessageStatsReport
# List attributes regarding all messages created
Report.report2 = CreatedMessagesReport
# Header, time, ID, size, hop count, delivery time, from, to
# remaining TTL, response, path
Report.report3 = DeliveredMessagesReport
# How long nodes were in range of each other
Report.report4 = ContactTimesReport
# Graphical representation of the network
Report.report5 = AdjacencyGraphvizReport
# Location (coordinates) of all messages reported at regular intervals
Report.report6 = MessageLocationReport

in this report are the number of dropped messages, the number of delivered messages, the delivery probability, the latencies, the overhead ratio and the hop count measurements.

The Message Delivery Report details when and how long each message takes to get from the source to destination as well as the path or route the message traversed. Table 3.3 is an excerpt of one message delivery report. The report contains a listing (in rows) for all delivered messages. In the report header, the *Time* is the time that the message was sent, the *MsgID* is tagged message identification number, the *Size* denotes the size of the message, *Hops* indicates the number of hops from source to destination, *Lat* refers to the latency, the time needed for a message

TABLE 3.2. Example Message Statistics Report: An example of a message statistics report produced from the ONE simulation.

Stat	Value
simtime:	3600.0260
created:	3600
started:	129442
relayed:	129441
aborted:	0
dropped:	0
removed:	129441
delivered:	3264
deliveryprob:	0.9067
responseprob:	0.0000
overheadratio:	38.6572
latencyavg:	214.9006
latencymed:	76.2370
hopcountavg:	35.2007
hopcountmed:	17
buffertimeavg:	6.3682
buffertimemed:	0.1480
rttavg:	NaN
rttmed:	NaN

to get from source to destination, *Src* and *Dst* are the source and destination of the message respectively, *TTL* is the remaining TTL of the packet at the time of delivery, *Rsp* indicates if a response was received and *Path* details the entire path of the message including source and destination.

TABLE 3.3. Example Message Delivery Report: An example of a message delivery report produced from the ONE simulation.

Time	MsgID	Size	Hops	Lat	Src	Dst	TTL	Rsp	Path
1.036	M1	100000	1	0.036	p75	p6	n/a	N	p75 p6
18.019	M18	100000	1	0.019	p62	p67	n/a	N	p62 p67
45.066	M45	100000	2	0.066	p32	p45	n/a	N	p32 p88 p45
52.096	M52	100000	3	0.096	p2	p9	n/a	N	p2 p5 p86 p9

3.3 Shortest Path Routing

Shortest Path is an semi-omnipotent router based upon Dijkstra's shortest path algorithm. It was created as the standard by which all simulated protocols are measured. For this protocol, the shortest path between every node and destination is calculated at each update interval and messages are routed accordingly. The shortest path is likely to change between update intervals which is why the shortest path is re-calculated each update period. The simulation results of *Shortest Path* represent the upper-bound on the delivery performance.

Shortest Path is not the optimum solution. It is limited to knowing the optimum state of the network at time t and not the optimum solution at all times $t+n$ where $n = [1, 2, 3, ...]$. The term semi-omnipotent in this case means, knowing all shortest paths in the network at this moment in time t.

3.4 Parameters

All thesis results were obtained using the same random way point simulation scenario, referred to as the *Random Scenario*. Each protocol was tested using the same set of buffer sizes and run 17 times with different random number generator (RNG) seeds to negate systematic simulation affects. Table 3.4 lists the simulation settings.

$3,600$ messages are created during the 1 hour simulation. Source and destination nodes are chosen randomly therefore each node is just as likely as any other to source or sink messages. The message time-to-live (TTL) is 300 minutes, explicitly set to be greater then the total simulation time so that TTL doesn't play a role in dropped messages. Messages are queued (but not necessarily transmitted) in FIFO order and only dropped due to queue overflow or protocol-based metrics.

The network map is 1 square kilometer, the radio broadcast range for all nodes is 100 meters, the message (packet) size is static at 100 kilobytes and there are 100

nodes that participate in the network simulation. The nodes can move randomly over the map at between 0.5 and 1.5 meters per second (walking speed) and once they've reached their target they will hold for anywhere between 0 and 1 second before continuing on to their next randomly chosen destination.

The nine different routing protocols that were tested and whose results are reported are:

- *Direct Delivery (DD)* [23]: Self-delivery.
- *Epidemic (EPI)* [8]: Packet flooding.
- *Prophet with Estimation (PRO)* [9]: Probabilistic routing.
- *MaxProp (MP)* [11]: Transmission and drop prioritization.
- *Backpressure LaB (LaB)*: Combination between Backpressure and the future position of the message in the neighbor's queue.
- *Spray and Wait (SaW)* [12]: Bounded multi-copy routing.
- *Shortest Path (SP)*: Omnipotent shortest path.
- *CASPaR (CASPaR)*: Single-copy, single-path.
- *CASPaR-MP*: Single-copy, multi-path.

TABLE 3.4. Simulation parameters as used by the Random Scenario simulations

Parameter Description	Value
World size	1km x 1km
Node count	100
Simulation Update Interval	0.037 seconds
Network packet rate	1 per second
Run time	3,600 seconds
Transmit speed	10 Mbps
Transmit range	100 meters
Buffer-sizes tested	.2, .5, 1, 3, 5, 10 and 30 MB
Send queue	FIFO queue
Node speed	0.5 - 1.5 meters per second
Node wait time	0.0 - 1.0 seconds
Message TTL	5 hours
Message period	1 second
Message size	100 KB
Node movement	RandomWayPoint
Movement warmup period	100 seconds
Map	Open map
Protocols tested	*Direct Delivery*, *Epidemic*, *PRoPHET*, *MaxProp*, *Spray and Wait*, *LaB Shortest Path*, *CASPaR* and *CASPaR-MP*
Queue Type	FIFO
Number of reports	6
Reports	Message Statistics, Created Messages Delivered Messages, Contact Times Adjacency Graph, Message Location
Message location granularity	60
MaxProp timescale	10
PRoPHET seconds in time unit	10
Spray and Wait number of copies	6
Spray and Wait mode	binary mode
CASPaR mode	single path OR multi-path
CASPaR minimum loop count	5

Chapter 4
Results

Here we present results from Random Scenario simulations focused on the performance metrics: Delivery Probability, Overhead Ratio, Hop Count and Packet Latency. Also reported are results from two additional investigations: 1) packet latency and 2) route distribution across the network. All performance metric plots, for all protocols except *MaxProp*, show 1-sigma uncertainty bars representing deviation between the 17 simulation runs. *MaxProp*'s simulation times prohibited multiple runs and therefore have no associated deviations.

4.1 Delivery Probability

Figure 4.1 relates buffer size and delivery probability. As buffer size increases so too does the delivery rate until a bounded maxima is reached. The maximum delivery rate for all protocols except *MaxProp* is reached at $>$ 10 MB buffer allowing for a good comparison between tested protocols. Results show four distinct protocol behaviors: 1) referred to as the *SP* group includes both *CASPaR* and *Shortest Path* routing protocols and is characterized by its steep rise in delivery probability settling close to or above 80 percent; 2) referred to as the *Direct* group includes *PRoPHET*, *Direct Delivery* and *LaB*. This group also has a relatively steep rise in delivery probability but settles at a much lower rate below 50 percent; 3) *Spray and Wait* is in a group by itself and can be identified by its slow rise, reaching a maximum at $>$ 10 MB buffer; 4) *MaxProp*, also in a group by itself, is unique. Its delivery probability maintains a shallow but constant rise reaching a maximum of 90 percent at a 30 MB buffer and still increasing.

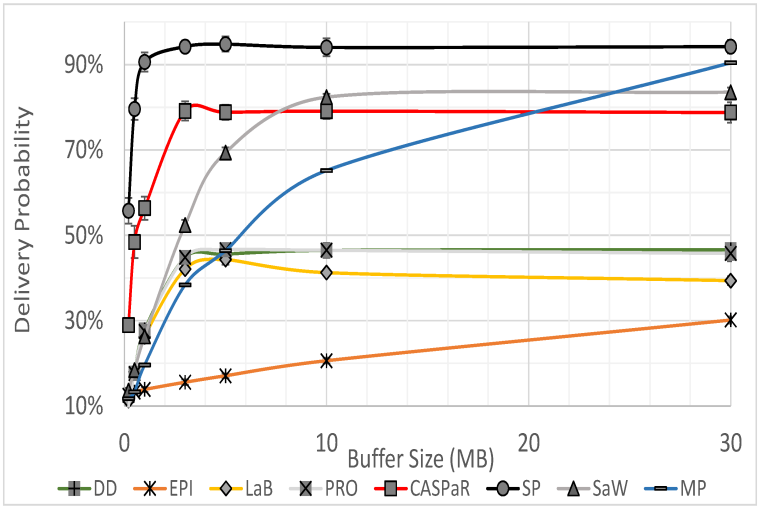


FIGURE 4.1. Delivery Probability: The delivery probability as a function of queue size for *Direct Delivery (DD)*, *Epidemic (EPI)*, *Backpressure (LaB)*, *PRoPHET (PRO)*, *CASPaR*, *Shortest Path (SP)*, *Spray and Wait (SaW)* and *MaxProp (MP)* routing.

Shortest Path sets the upper-bound on the delivery rate at 95 percent. *Direct Delivery* sets the *effective* lower bound at 45 percent due to its simplistic routing scheme, *hold packets until target destinations are met*, even though there are several protocols that don't perform as well. This result reveals that any two nodes in the network are in contact with each other 45 percent of the time. All protocols should be capable of at least matching this delivery rate.

CASPaR delivers 55 percent or more of its packets using a buffer of only 1 MB. This is twice the number of packets delivered by the next best algorithm revealing that *CASPaR* either delivers packets more quickly or they are being more evenly distributed across the network or both. Alternatively, *Spray and Wait* performs poorly until its queue size reaches $>$ 10 MB and then barely outperforms *CASPaR* while *MaxProp* starts poorly but outperforms all but *Shortest Path* once a $>$ 25 MB buffer is employed.

4.2 Latency

Average latency, defined by the ONE simulator as the average amount of time it takes all delivered packets to travel from source to destination, may be the most

meaningful metric of all since it provides not only the rate of packet delivery, but also an indirect performance metric for delivery probability. However, average latency can be falsely biased since only those packets that reach their destinations contribute to the reported average latency. It is these undelivered packets that if delivered would raise the average latency. Notice that Figure 4.2 shows low average latencies at small buffer sizes but poor delivery performance. Comparable latency measurements must be obtained when the buffer size is large enough so that packet drop is not a factor. Figure 4.2 shows this to be at < 10 MB for all protocols except *MaxProp* and *Epidemic*. Regardless, the median latency, as shown in Figure 4.3, must be used to gain a better approximation of true latency since the average can also be biased by extremely large latencies.

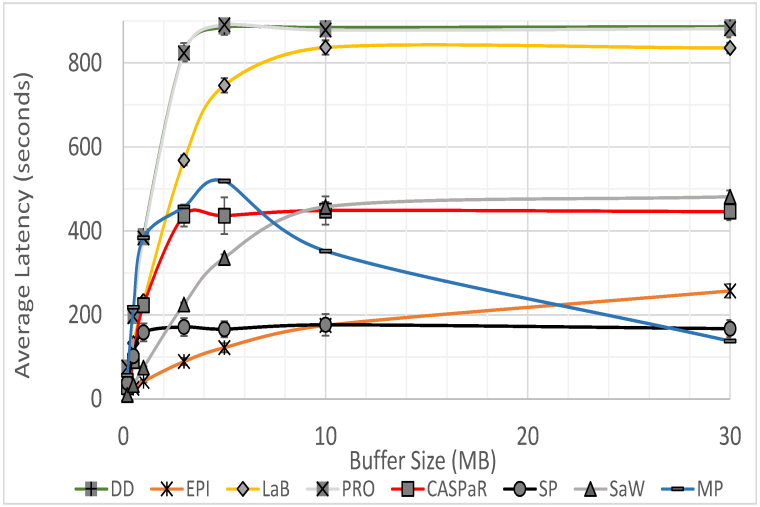


FIGURE 4.2. Average Latency: The average end-to-end packet latency as a function of queue size for *Direct Delivery (DD)*, *Epidemic (EPI)*, *Backpressure (LaB)*, *PRoPHET (PRO)*, *CASPaR, Shortest Path (SP)*, *Spray and Wait (SaW)* and *MaxProp (MP)* routing.

To illustrate, notice the median latency for *CASPaR* is nearly half its average and for *Shortest Path*, it is nearly a third, an indication that there are low-latency measurements skewing the average. The median and average latencies of *MaxProp*,

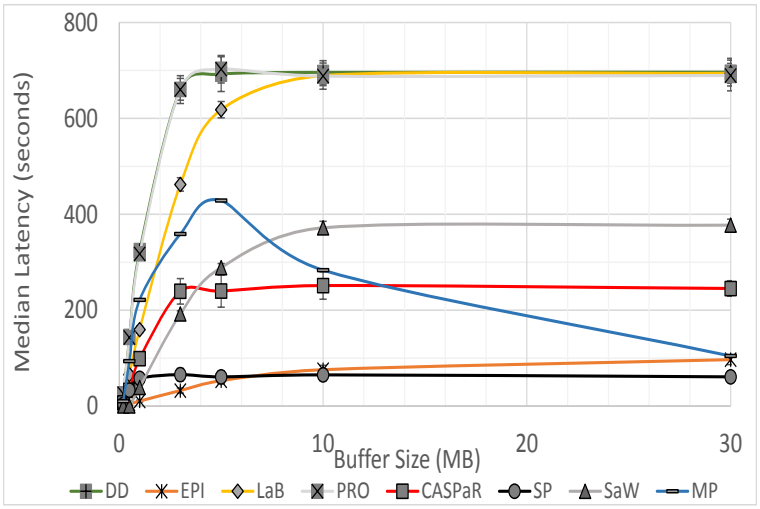


FIGURE 4.3. Median Latency: The median latency as function of queue size for *Direct Delivery (DD)*, *Epidemic (EPI)*, *Backpressure (LaB)*, *PRoPHET (PRO)*, *CASPaR*, *Shortest Path (SP)*, *Spray and Wait (SaW)* and *MaxProp (MP)* routing.

Spray and Wait and the protocols in the *Direct* group are similar in value indicating low and high latency measurements are more balanced for these protocols.

Figure 4.3 shows that *CASPaR* performs quite well with a median latency of about 250 seconds. *MaxProp* exhibits unique behaviour as it rises above 400 seconds at a 5 MB buffer but then drops to below a 100 seconds at a 30 MB buffer. It is just above *CASPaR*'s 10 MB buffer latency of 300 seconds. (*Epidemic* isn't included in the comparison due to its extremely low delivery probability) and high latency.

Figure 4.4 presents a more in-depth latency analysis for 10 MB protocol buffers. The comparison includes protocols: *Shortest Path*, *CASPaR*, *Spray and Wait*, *Direct Delivery* and *MaxProp*. The frequencies have been normalized so that a direct comparison can be made but it should be noted that the actual total count for *MaxProp* is only about $2,300$ compared with approximately $50,000$ for the others. Also provided is evidence as to why *Shortest Path* performs so well comparatively. It delivers many more packets in the < 1 second range and far fewer in the > 512 second range. *CASPaR* is the only other protocol which consistently performs well at the latency extremes.

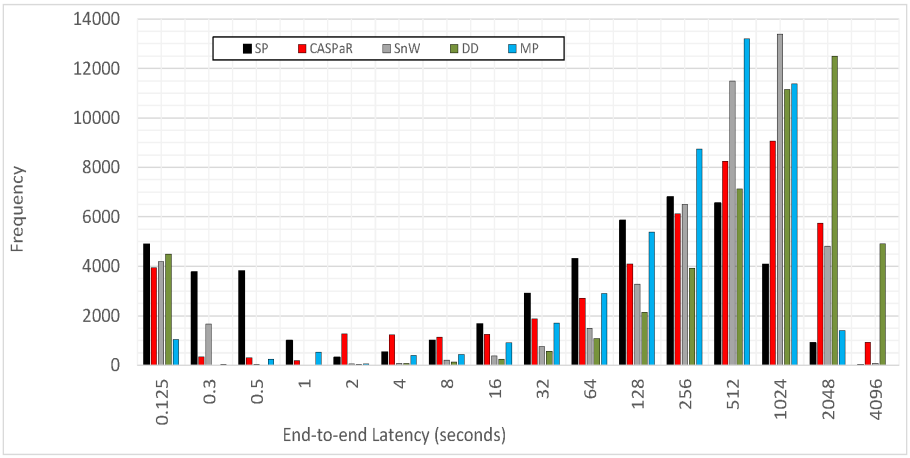


FIGURE 4.4. Latency Frequency Distribution: The frequency of latency distributions for the experimental results of *Direct Delivery (DD)*, *Spray and Wait (SaW)*, *Shortest Path (SP)*, *CASPaR* and *MaxProp (MP)* protocols. All 17 runs for each is included in the analysis. The bin size is in log base 2 format to accentuate lower latencies.

A closer look uncovers protocol and simulation behavior. For example, all protocols, with the exception of *MaxProp* deliver a proportionately high number of packets in the 0.125 second bin indicating the likelihood that source and destination nodes are within a 2-hop range at the time of packet creation. The frequency of delivered messages in the $0.125 - 1$ second bins drops quickly for all protocols except *Shortest Path*. This may reveal the existence of multi-hop connected paths at the time of packet creation and provide a multi-hop latency measurement of < 1 second. The > 1 second bins are most likely a convoluted measure of the average length of time routes remain disconnected as well as protocols ability to move packets closer to their destination across disconnected paths. If so then *Shortest Path* provides a good performance indicator and comparison tool. Table 4.1 shows that *Shortest Path* delivers more packets at latencies of < 128 seconds and less packets at latencies > 256 seconds than all protocols except for *CASPaR* which performs better in the $1 - 8$ second latency range and almost as well in the $8 - 256$ latency range. This shows why *Shortest Path* preforms better than all other protocols and why *CASPaR* performs almost as well when using a 10 MB buffer.

TABLE 4.1. Direct latency comparisons ratios of *Spray and Wait (SaW)*, *CASPaR*, and *MaxProp (MP)* vs. *Shortest Path*

Latency Bin	$\frac{SP}{SaW}$	$\frac{SP}{CASPaR}$	$\frac{SP}{DD}$	$\frac{SP}{MP}$
$0.0 - 0.125$	1.2	1.2	1.1	4.7
$0.125 - 0.25$	2.3	11.2	1255.0	91.8
$0.25 - 0.5$	131.5	12.6	1271.3	15.4
$0.5 - 1.0$	41.8	5.3	100.3	1.9
$1.0 - 2.0$	6.4	0.2	8.8	5.1
$2.0 - 4.0$	6.8	0.4	7.5	1.3
$4.0 - 8.0$	4.9	0.9	7.2	2.3
$8.0 - 16.0$	4.5	1.3	6.6	1.8
$16.0 - 32.0$	3.8	1.5	5.1	1.7
$32.0 - 64.0$	2.9	1.6	4.0	1.5
$64.0 - 128.0$	1.8	1.4	2.7	1.1
$128.0 - 256.0$	1.0	1.1	1.7	0.8
$256.0 - 512.0$	0.6	0.8	0.9	0.5
$512.0 - 1024.0$	0.3	0.4	0.4	0.4
$1024.0 - 2048.0$	0.2	0.2	0.1	0.6
$2048.0 - 4096.0$	0.2	0.0	0.0	0.0

4.3 Overhead

The overhead ratio is proportional to a protocol's energy expenditure and is defined by the ONE simulator to be $O_r(t) = (P_r(t) - P_d(t))/P_d(t)$ where P_r is the total number of packets relayed by time t and P_d is the total number of packets delivered by time t. Overhead ratio is an important performance metric for low-power DTN devices which typically do not have energy to spare and where the goal is to deliver packets to their destinations in the most energy efficient means possible. The following protocol's overhead results are not shown in Figure 4.5: 1) *MaxProp* - its overhead is > 1700, 2) *Direct Delivery* - its overhead is always 0, 3) *PRoPHET* - its overhead is approximately 6 but has a high standard deviation of about $(+/-12)$ and 4) *Epidemic* - its overhead is > 4500.

Since *CASPaR* does not replicate packets, the overhead is directly proportional to the number of packet hops. Figure 4.5 shows that the *CASPaR* maintains an

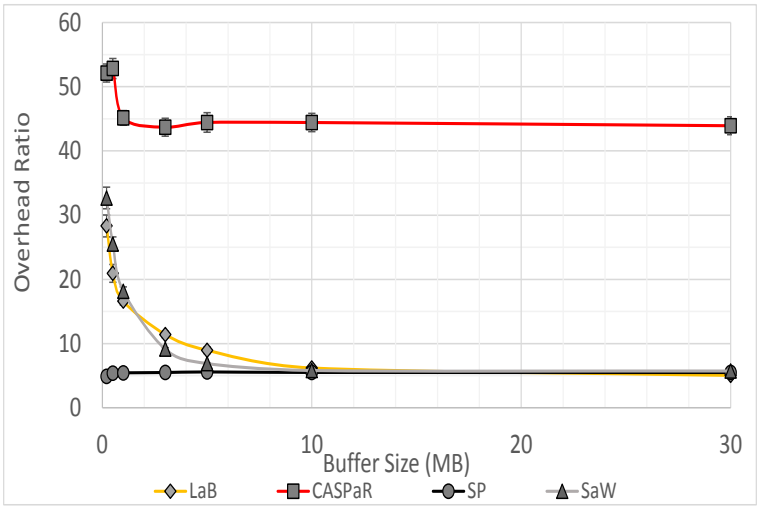


FIGURE 4.5. Overhead Ratio: The overhead ratio required to transfer a packet from source to destination as a function of buffer size for *Backpressure (LaB)*, *CASPaR*, *Shortest Path (SP)* and *Spray and Wait (SaW)* routing.

overhead ratio of approximately 40 while *Shortest Path*, *LaB*, and *Spray and Wait* all maintain overhead ratios of about 5. Figure 4.6 shows that *CASPaR's* overhead ratio can reduced at the expense of latency and delivery probability and vice versa.

4.4 Hop Count

Hop count, defined as the number of nodes a packet traverses from source to destination. The final transfer to the destination node isn't considered a hop and therefore *Direct Delivery*'s hop count is always 0. Figure 4.7 shows the average number of packet hops in the protocols tested. It isn't surprising that the hop count and overhead are similar for *CASPaR* as well as *Shortest Path* since they are single-copy protocols. The overhead results of *Shortest Path* indicate the optimum number of average hops, to be about 6.

It is clear from results reported here that the stated goal of minimizing latency and maximizing delivery can not be met without compromise. For example, *Direct Delivery* has 0 overhead but performs poorly in regards to latency and delivery probability. Alternatively, *MaxProp* delivers a high percentage of its packets at relatively low latencies but requires a much larger buffer and very high overhead to do

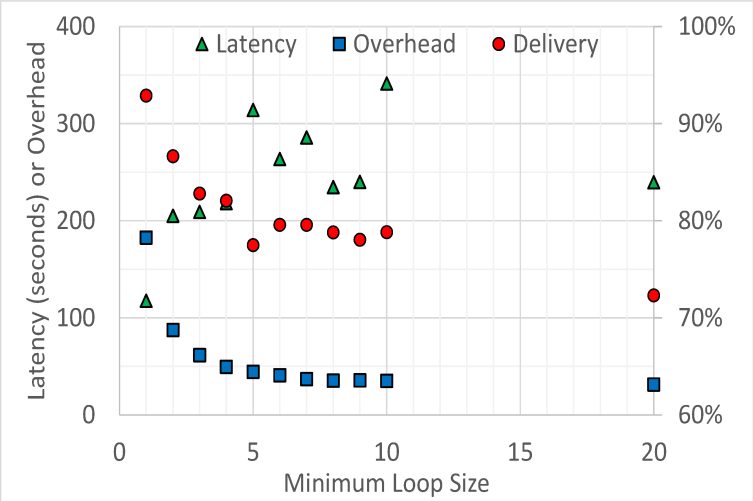


FIGURE 4.6. Affect Minimum Loop Size has on Performance: As the minimum loop size decreases (the number of nodes required in routing path before looping back to itself) the overhead increases but the median (as shown in this figure) and average latencies decrease while delivery probability increases.

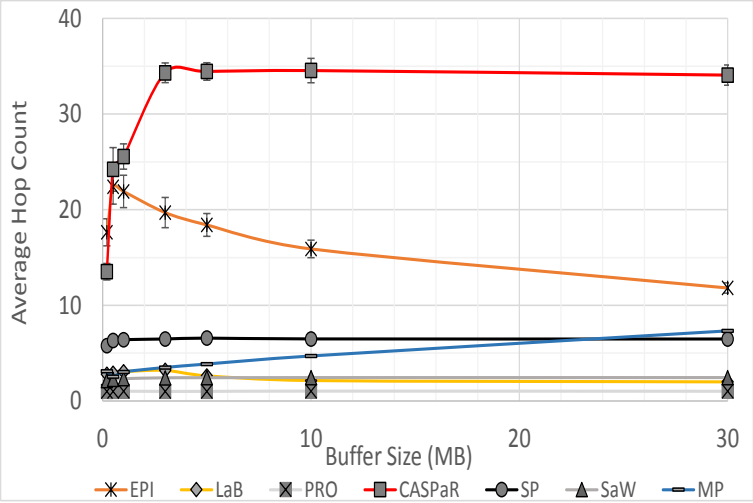


FIGURE 4.7. Average Hop Count: The average number of nodes a packet traverses from source to destination as a function of buffer size for *Epidemic (EPI)*, *Backpressure (LaB)*, *PRoPHET (PRO)*, *CASPaR*, *Shortest Path (SP)*, *Spray and Wait (SaW)* and *MaxProp (MP)* routing. The transfer to the destination node is not considered as a hop.

so. *CASPaR* is capable of out-performing tested DTN protocols while maintaining relatively low overhead.

4.5 Load Balancing

Presumably, given a homogeneous set of packet destinations, the more equally packets are distributed across a network, the more efficiently it will function at high loads. There are many factors that contribute to this such as the variation in randomly chosen source and destination nodes. Other factors such as graph connectivity play a role as well.

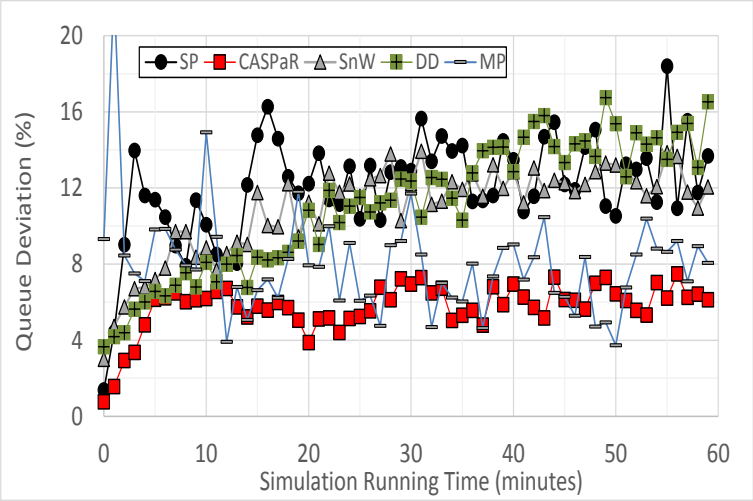


FIGURE 4.8. Queue Size Deviation: Queue deviation integrated over 1 minute periods for all 60 minutes of the simulation for *Shortest Path*, *CASPaR*, *Spray and Wait*, *Direct Delivery* and *MaxProp*.

Figure 4.8 shows average queue deviation for *Shortest Path*, *CASPaR*, *Spray and Wait*, *Direct Delivery* and *MaxProp*. This and Table 4.2 show that *CASPaR* more evenly distribute packets over the network (load balance) than compared protocols. The variation across queues in *CASPaR* is half that of *Direct Delivery* and *Spray and Wait* and a bit lower and tighter than *MaxProp*.

However, *Shortest Path* experiences the largest variation and yet by every metric, it out-performs all protocols. This indicates that either high-performance does not

TABLE 4.2. Average Queue Deviation: The average deviation (as a percentage) between queues across the network. The queue sizes are integrated over 1 minute periods and those are averaged together to get the average deviation over the entire simulation for *Shortest Path (SP)*, *CASPaR*, *Spray and Wait (SaW)*, *Direct Delivery (DD)* and *MaxProp (MP)* routing.

SP	*CASPaR*	*SaW*	*DD*	*MP*
12.08	5.75	10.93	11.08	8.00

depend upon an even distribution of packets across queues or that the network load applied during testing (1 packet per second) wasn't heavy enough to highlight the property.

4.6 Single Path vs. Multi-path

The results from the multi-path CASPaR variant (*CASPaR-MP* are reported in this section. The same performance metrics: Delivery Probability, Overhead Ratio, Hop Count and Packet Latency are used to compare *CASPaR-MP* with *CASPaR* as well as *Shortest Path*, *Spray and Wait* and *MaxProp*. Again, all protocols including *CASPaR-MP* but excluding emphMaxProp were run 17 times with different RNG seeds to account for statistical variations in the simulations. The standard deviations are shown as 1-sigma error bars in the primary performance metric plots.

4.6.1 Delivery Probability

As shown in Figure 4.9, *CASPaR-MP* performs slightly better than *CASPaR* and about as well as *Spray and Wait* in delivery probability when a $>$ 10 MB buffer is used. The delivery behavior is almost identical to that of *CASPaR* in that the curves a function of buffer size follow each other almost perfectly. This is not unexpected since the two protocol algorithms are so similar.

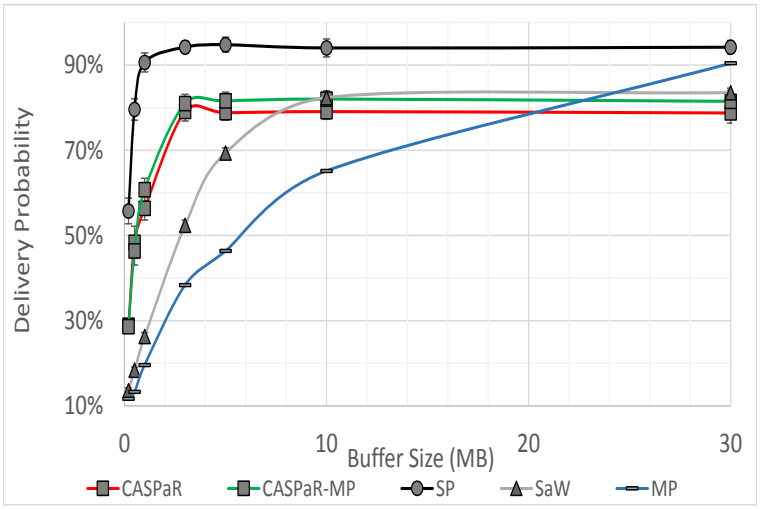


FIGURE 4.9. Delivery Probability - Single vs. Multi-path: The delivery probability as a function of queue size for *CASPaR*, *CASPaR-MP*, *Shortest Path (SP)*, *Spray and Wait (SaW)* and *MaxProp (MP)* routing.

4.6.2 Latency

Figures 4.10 and 4.11 show that *CASPaR-MP* performs about 10 percent better in both average and median latencies and almost breaks the 200 second median latency barrier.

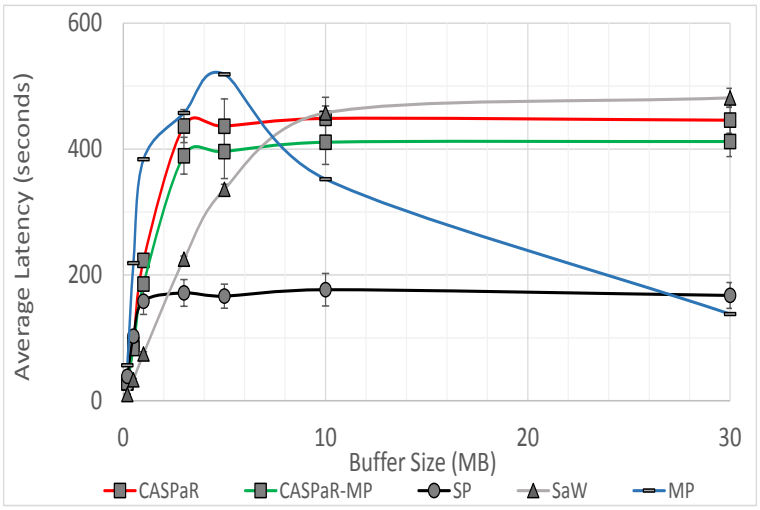


FIGURE 4.10. Average Latency - Single vs. Multi-path: The average end-to-end packet latency as a function of queue size for *CASPaR*, *CASPaR-MP*, *Shortest Path (SP)*, *Spray and Wait (SaW)* and *MaxProp (MP)* routing.

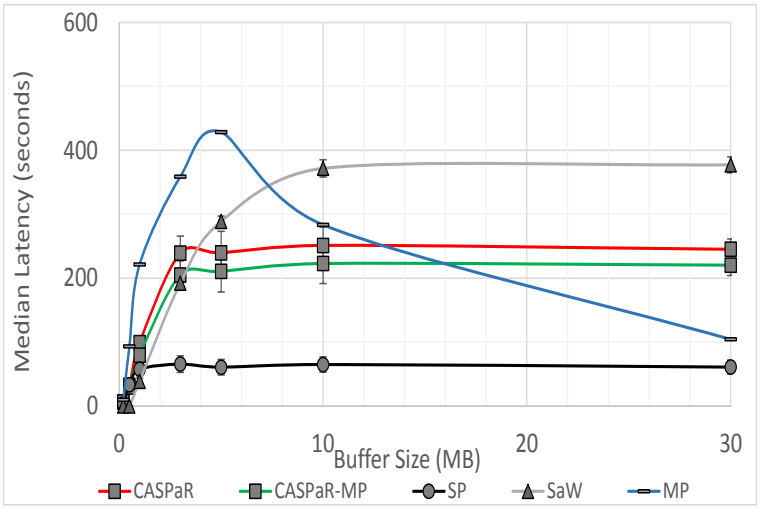


FIGURE 4.11. Median Latency - Single vs. Multi-path: The median latency as function of queue size for *CASPaR*, *CASPaR-MP*, *Shortest Path (SP)*, *Spray and Wait (SaW)* and *MaxProp (MP)* routing.

Figure 4.12 provides insight into why *CASPaR-MP*'s latency is a bit better. Notice that in the low and high latency bins *CASPaR-MP* slightly outperforms *CASPaR* meaning *CASPaR-MP*, generally delivers more packets in the low latency bins and less packets in the higher latency bins.

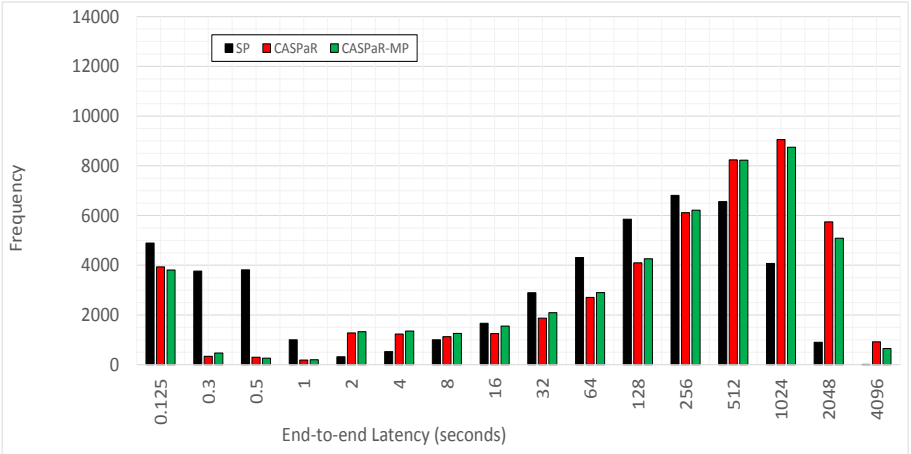


FIGURE 4.12. Latency Frequency Distribution - Single vs. Multi-path: The frequency of latency distributions for the experimental results of *Direct Delivery (DD)*, *Spray and Wait (SaW)*, *Shortest Path (SP)*, *CASPaR* and *MaxProp (MP)* protocols. All 17 runs for each is included in the analysis. The bin size is in log base 2 format to accentuate lower latencies.

This is made even more clear in Table 4.3 where it shown that *CASPaR* slightly outperforms *CASPaR-MP* in the $0.0 - 0.125$ and $0.25 - 0.5$ latency bins but *CASPaR-MP* outperforms *CASPaR* in all others.

TABLE 4.3. Multi-path Latency Ratios: Direct latency comparisons ratios of *CASPaR* vs. *CASPaR-MP*

Latency Bin	$\frac{CASPaR}{CASPaR-MP}$
$0.0 - 0.125$	0.97
$0.125 - 0.25$	1.39
$0.25 - 0.5$	0.86
$0.5 - 1.0$	1.04
$1.0 - 2.0$	1.04
$2.0 - 4.0$	1.10
$4.0 - 8.0$	1.12
$8.0 - 16.0$	1.24
$16.0 - 32.0$	1.11
$32.0 - 64.0$	1.07
$64.0 - 128.0$	1.04
$128.0 - 256.0$	1.02
$256.0 - 512.0$	1.00
$512.0 - 1024.0$	0.97
$1024.0 - 2048.0$	0.89
$2048.0 - 4096.0$	0.71

4.6.3 Hop Count and Overhead

When comparing the single-path and multi-path variants, delivery probability and latency is quite important but it might be more telling that *CASPaR-MP* has a lower overhead and hop count. All of the results including delivery probability, latency, overhead and hop-count, taken as a whole, indicate that the multi-path approach is promising. Latency is decreases while hop count and overhead decreases as well. This is precisely the stated goal of *CASPaR*, to minimize latency, maximize delivery probability and avoid congestion.

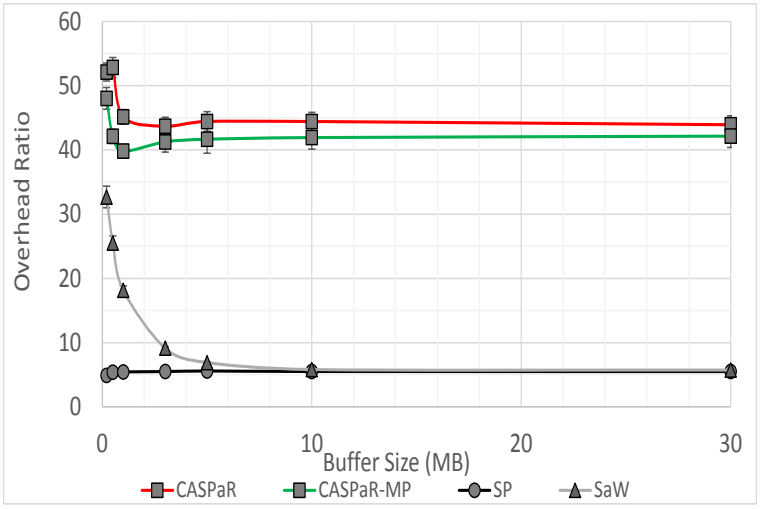


FIGURE 4.13. Overhead Ratio - Single vs. Multi-path: The overhead ratio required to transfer a packet from source to destination as a function of buffer size for *CASPaR*, *CASPaR-MP*, *Shortest Path (SP)* and *Spray and Wait (SaW)* routing.

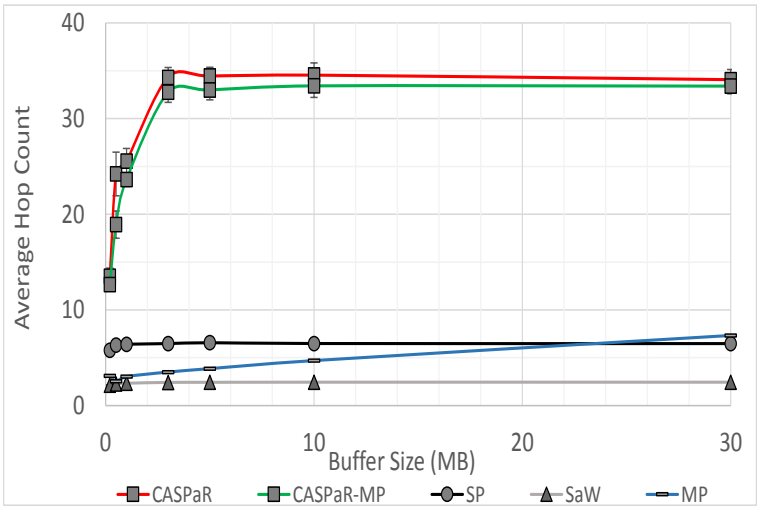


FIGURE 4.14. Average Hop Count - Single vs. Multi-path: The average number of nodes a packet traverses from source to destination as a function of buffer size for *CASPaR*, *CASPaR-MP*, *Shortest Path (SP)*, *Spray and Wait (SaW)* and *MaxProp (MP)* routing. The transfer to the destination node is not considered as a hop.

4.7 Summary

An overview of the statistical results of the simulation tests are provided in Figure 4.15 making comparison between protocol performance easy. The results as written to the figure are approximate.

Protocol	Delivery Probability (%)	Median Latency (seconds)	Overhead	Hop Count
Direct Delivery	45	700	0	0
Epidemic	20	75	4500	15
Back pressure	40	700	5	<5
PRoPHETv2	45	700	6	<5
MaxProp	65	300	1700	5
Spray and Wait	70	375	5	<5
Shortest Path	95	75	5	6
CASPaR (MLS=5)	80	250	45	45
CASPaR-MP	82	220	-	-
CASPaR- (MLS=1)	92	120	180	similar to overhead

FIGURE 4.15. Result Summary Table: A summary of the protocol simulation results shown for the 10 MB buffer size for delivery probability, median latency, overhead and hop count. Also shown are the results for *CASPaR* with the minimum loop size (MLS) set to 1. It should be noted that *MaxProp* continues to perform better as the buffer size increases whereas all other protocols peak at 10 MB.

Chapter 5
Conclusion

5.1 Summary

We have developed an extensible protocol, one that doesn't depend upon mobility predictions or data mules. A protocol that can handle a relatively heavy network-load using small network resources and by all measurements, one that should be able to handle an even heavier network load than applied during these simulations. We have shown that *CASPaR* improves network performance and while *Spray and Wait* and *MaxProp* also perform well under the same experimental conditions, both require much larger buffers and in the case of *MaxProp*, a much larger overhead.

5.2 Future Study

Preliminary results have proven *CASPaR* to be effective and further testing is required in order to better quantify its capabilities and undoubtedly prove how effectively it routes packets and avoids congestion. The tests performed here were limited to the Random Scenario simulation as described previously and were not able to explicitly show how well *CASPaR* routes and avoids congestion.

In order to test for routing performance, specific simulations must accentuate the routing portion of *CASPaR* while minimizing the affect of the congestion avoidance portion of its algorithm. This can be done by lowering the event (packet) rate so that packet congestion is minimized. The simulation results should be analyzed for individual path deviations from the shortest path truth table for each packet hop. The number of times the wrong decision was made should be compared to the number of times the correct decision was made accounting for degrees of freedom.

Once routing performance is understood, *CASPaR*'s ability to avoid congestion must be investigated. This test should be easier to perform than its predecessor

and can be accomplished using the simplistic Random Scenario simulation and incrementally dialing up the event rate until a drop in delivery performance is observed. This inflection point will be the boundary between unhindered and congested packet flow. The event rate should continue to be increased until either some steady-state is observed or until failure (the point at which very few or no packets are delivered). Once this occurs, the results should be thoroughly investigated looking for specific signs leading to the cause of failure and specifically, the aspect of even packet distributions.

Variations of the *CASPaR* protocol were developed that consider multiple routes from a node in the transmission cost calculation as opposed to just a single one. This variant out-performed the one presented here and should be further explored. The same two tests (routing and congestion) can be applied to the multi-path *CASPaR* variant. The results of which should be compared with the standard variant's. The next logical step would be to experiment with breaking packets at the source and rebuilding them at the destination and whether or not this provides any performance increase.

Finally, it is important that *CASPaR* is proven, analytically, to be throughput optimal such that it has the ability to support the maximum throughput that is queue-able as defined by [24], [25]. However, it should be shown to be so at high loads proving that *CASPaR* is in fact a back-pressure algorithm when stressed and an historical routing algorithm when not stressed.

References

[1] C.E. Perkins and E.M. Royer. Ad-hoc on-demand distance vector routing. In *IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, 1999.

[2] P. Jacquet, P. Muhlethaler, T. Clausen, A. Laouiti, A. Qayyum, and L. Viennot. Optimized link state routing protocol for ad hoc networks. In *Proceedings of IEEE INMIC 2001*, pages 62–68, 2001.

[3] Z. J. Haas, M. R. Pearlman, and P. Samar. The zone routing protocol (zrp) for ad hoc networks. IETF Draft, 2002.

[4] R. J. D'Souza and J. Jose. Routing Approaches in Delay Tolerant Networks: A Survey. *International Journal of Computer Applications*, 1(17):8–14, February 2010.

[5] Mengjuan Liu, Yan Yang, and Zhiguang Qin. A survey of routing protocols and simulations in delay-tolerant networks. In Yu Cheng, DoYoung Eun, Zhiguang Qin, Min Song, and Kai Xing, editors, *Wireless Algorithms, Systems, and Applications*, volume 6843 of *Lecture Notes in Computer Science*, pages 243–253. Springer Berlin Heidelberg, 2011.

[6] P. Hui, J. Crowcroft, and E. Yoneki. BUBBLE Rap: Social-Based Forwarding in Delay-Tolerant Networks. *IEEE Transactions on Mobile Computing*, 10(11):1576–1589, Nov 2011.

[7] Thrasyvoulos Spyropoulos, K. Psounis, and C.S. Raghavendra. Single-copy routing in intermittently connected mobile networks. In *Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004. 2004 First Annual IEEE Communications Society Conference on*, pages 235–244, Oct 2004.

[8] A. Vahdat and D. Becker. Epidemic routing for partially-connected ad hoc networks. Technical report, Duke Tech Report CS-2000-06, 2000.

[9] A. Lindgren, A. Doria, and O. Scheln. Probabilistic routing in intermittently connected networks. In *ACM SIGMOBILE Mobile Computing and Communications Review*, pages 19–20, 2003.

[10] M. Naziruddin and M. Pushpalatha. A Dynamic Approach To History Based DTN Routing on Delivery Predictabilities. *International Journal of Applied Engineering Research*, 10(7):17275–17282, 2015.

[11] J. Burgess, B. Gallagher, and D. Jensen. Maxprop: Routing for vehicle-based disruption-tolerant networks. In *Proceedings of IEEE Infocom, April 2006*, 2006.

[12] T. Spyropoulos, K. Psounis, and C Raghavendra. Spray and wait: an efficient routing scheme for intermittently connected mobile networks. In *Proceedings of the 2005 ACM SIGCOMM workshop on Delay-tolerant networking*, pages 252–259, 2005.

[13] S. Moeller, A. Sridharan, B. Krishnamachari, and O. Gnawali. Routing without routes: the backpressure collection protocol. In *IPSN*, pages 279–290, 2010.

[14] A. Dvir and A V. Vasilakos. Backpressure-based routing protocol for dtns. In *SIGCOMM*, pages 405–406, 2010.

[15] J. Ryu, L. Ying, and S. Shakkottai. Back-pressure routing for intermittently connected networks. In *INFOCOM*, pages 1–5, 2010.

[16] M. Alresaini, M. Sathiamoorthy, B. Krishnamachari, and M. J. Neely. Backpressure with Adaptive Redundancy (BWAR). In *INFOCOM*, pages 2300–2308, FL, USA, March 2012.

[17] A. P. Silva, S. Burleigh, C. M. Hirata, and K. Obraczka. A survey on congestion control for delay and disruption tolerant networks. *Ad Hoc Network*, 25(1):480–494, Aug. 2015.

[18] Juan-Carlos Cano Sergio M. Tornell, Carlos T. Calafate and Pietro Manzoni. Dtn protocols for vehicular networks: An application oriented overview. *Communications Surveys and Tutorials*, 17(2):868–887, 2015.

[19] K. Fall. A delay-tolerant network architecture for challenged internets. In *Proceedings of the SIGCOMM '03 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 27–34, 2003.

[20] Brian Dunbar. Nasa's space cubes: Small satellites provide big payoffs, Sept 2015.

[21] Google. Loon for all, balloon-powered internet for everyone @ONLINE, October 2015.

[22] A. Keränen, J. Ott, and T. Kärkkäinen. The one simulator for dtn protocol evaluation. In *International Conference on Simulation Tools and Techniques*, pages 1–10, 2009.

[23] R. S. Mangrulkar and M. Atique. Performance evaluation of flooding based delay tolerant routing protocols. *International Journal of Computer Applications*, pages 35–40, February 2012.

[24] Leandros Tassiulas and Anthony Ephremides. Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks. *IEEE Transactions on Automatic Control*, 37:1936–1948, 1992.

[25] K. Jagannathan, M. Markakis, E. Modiano, and J.N. Tsitsiklis. Throughput optimal scheduling in the presence of heavy-tailed traffic. In *Communication, Control, and Computing (Allerton), 2010 48th Annual Allerton Conference on*, pages 953–960, Sept 2010.

Appendix A: Simulation Code

```
package routing;

import java.util.*;

import core.Connection;
import core.DTNHost;
import core.Message;
import core.Settings;
import core.SimClock;


public class SelfishRouter_v7 extends ActiveRouter {

    /**
     * Selfish Router Properties
     */

    /** Question :: How does the manipulation of the automatic request for bid period change the behavior of the routing
        protocol? */

    public static final boolean   TRACE_UPDATE = true;
    public static final boolean   TRACE_RX = false;
    public static final boolean   TRACE_TX = false;
    public static final boolean   TRACE_CREATE = false;
    public static final boolean   TRACE_DST_MAP = false;
    public static final boolean   TRACE_RCVD_BID = false;
    public static final boolean   TRACE_UPDATE_Cs = false;
    public static final boolean   TRACE_CHEAP_DST = false;

    public static final int       N_NODES = 100;
    public static final double    AUTO_RFB_PERIOD = 1.0;    // upon expiration, a 'Request For Bids' is broadcast

    public static final int       MSG_TTL = -1;             // -1 indicates an infinite TTL setting
    public static final double    Cc_INIT = 10.0;
    public static final double    Cs_INIT = 0.0;
    public static final double    MAX_BID = 10000.0;

    public static final String    SELFISH_NS = "SelfishRouter" // Selfish router's setting namespace ({@value})

    private int []            relayAddress  = new int[N_NODES];
    private int []            pRelayAddress = new int[N_NODES];
    private double[]          B = new double[N_NODES];
    private double[]          Bs = new double[N_NODES];         // this is the winning bid, the lowest bid or transmission
        cost of all neighbors
    private double[]          Bm = new double[N_NODES];         // this is the winning bid, the lowest bid or transmission
        cost of all neighbors
    private double[]          minNeighborBid = new double[N_NODES];
    private double[]          T = new double[N_NODES];        // the transmission cost, the cost to directly transfer a packet
        from this node to its destination
    private double[]          theta = new double[N_NODES];      // some measure of the proximity of two nodes. Upon each
        update(),
                                                              // the connected check count is incremented and all nodes
                                                                  are checked
                                                              // whether they are in-range of other nodes. If so then the
                                                                  connected count
                                                              // is set equal to the connected checks. If not then only
                                                                  the connected
                                                              // check count is incremented.

    private double[]          Q = new double[N_NODES];          // sum of the amount of time all packets destined for some
                                                                // node @ time (t) have been resident in node (n)

    private int []            connectedChecks;
    private int []            connectedCount;
    private int []            pktCount;
    private boolean[]         connected;
    private boolean[]         inRange = new boolean[N_NODES];

    private double            lastCcUpdateTime;
    private double            lastBidUpdateTime;
    private int               updateCount;

    /** SelfishRouter's settings name space ({@value})*/
    public static final String MULTIPATH_MODE = "multiPathMode";
    public static final String SET_NODE_LOOPCOUNT = "nodeLoopCount";
    /** SelfishRouter's settings name space ({@value})*/
    public static final String SELFISHROUTER_NS = "SelfishRouter";
    /** Message property key */
    public static final String MSG_COUNT_PROPERTY = SELFISHROUTER_NS + "." +
       "copies";

    protected boolean isMultipath;
    protected boolean isFreeroute;
```

```
    protected int      nodeLoopCount;

    /**
     * Constructor. Creates a new message router based on the settings in
     * the given Settings object.
     * @param selfishRouter_vc2_5 The settings object
     *
     */
    public SelfishRouter_v7( Settings s  ) {
        super(s);

        Settings snwSettings = new Settings(SELFISHROUTER_NS);
        isMultipath = snwSettings.getBoolean(MULTIPATH_MODE);
        nodeLoopCount = snwSettings.getInt(SET_NODE_LOOPCOUNT);

        this.connectedChecks = new int[N_NODES];
        this.connectedCount = new int[N_NODES];
        this.connected = new boolean[N_NODES];
        this.pktCount = new int[N_NODES];
        this.lastCcUpdateTime = SimClock.getTime();
        this.updateCount = 0;

        for (int i=0; i<N_NODES; i++) {
            this.relayAddress[i] = -1;
            this.pRelayAddress[i] = -1;
            this.minNeighborBid[i] = MAX_BID;
            this.B[i] = MAX_BID;
            this.Bs[i] = MAX_BID;
            this.Bm[i] = MAX_BID;
            this.T[i] = Cc_INIT;
            this.theta[i] = Cs_INIT;
            this.inRange[i] = false;
            this.connectedChecks[i] = 0;
            this.connectedCount[i] = 0;
            this.connected[i] = false;
            this.pktCount[i] = 0;
        }
    }


    /**
     * Copy constructor.
     * @param r The router prototype where setting values are copied from
     */
    protected SelfishRouter_v7( SelfishRouter_v7 r ) {
        super(r);

        this.isMultipath = r.isMultipath;
        this.nodeLoopCount = r.nodeLoopCount;

        this.connectedChecks = new int[N_NODES];
        this.connectedCount = new int[N_NODES];
        this.connected = new boolean[N_NODES];
        this.pktCount = new int[N_NODES];
        this.lastCcUpdateTime = SimClock.getTime();
        this.updateCount = 0;

        for (int i=0; i<N_NODES; i++) {
            this.relayAddress[i] = -1;
            this.pRelayAddress[i] = -1;
            this.minNeighborBid[i] = MAX_BID;
            this.B[i] = MAX_BID;
            this.Bs[i] = MAX_BID;
            this.Bm[i] = MAX_BID;
            this.T[i] = Cc_INIT;
            this.theta[i] = Cs_INIT;
            this.inRange[i] = false;
            this.connectedChecks[i] = 0;
            this.connectedCount[i] = 0;
            this.pktCount[i] = 0;
        }
    }


    /**
     * Update is called once per simulation tick.  Within the update() function we have to:
     * -
     */
    @Override
    public void update() {
        super.update();

        this.updateCount++;

        if (this.lastBidUpdateTime + AUTO_RFB_PERIOD < SimClock.getTime()) {
            updateDestinationBids( );                    // update all bids for all connected hosts to all connections they have
                routes to

            if (TRACE_UPDATE) if (this.updateCount >= 10000) {
//              printDstTable( );
                this.updateCount = 0;
```

```
        }
    }

    if (!canStartTransfer() || isTransferring()) {
        return;                                     // nothing to transfer or is currently transferring
    }

    if (exchangeDeliverableMessages() != null) {        // try messages that could be delivered to final recipient
        return;
    }

    transmitCheapestOldestMessage( );
}


@Override
protected int checkReceiving(Message m) {
    int recvCheck = super.checkReceiving(m);

    if (recvCheck == RCV_OK) {
        /* don't accept a message that has already traversed this node */
        int fromIndex = m.getHops().size() - this.nodeLoopCount;
        int toIndex = m.getHops().size();

        if (fromIndex < 0) fromIndex = 0;
        if (fromIndex > toIndex) toIndex = fromIndex;

        if (m.getHops().subList(fromIndex, toIndex).contains(getHost())) {
            recvCheck = DENIED_OLD;
        }
    }

    return recvCheck;
}


/**
 * Receive message is called when another host sends this host a message. In addition to what is done in
 * the receiveMessage() function in the ActiveRouter class and in the MessageRouter class, the SelfishRouter
 * has to check to see if the message's destination is already in the destination hash map already. If not
 * it has to be put into the destination hash map and initialized.
 */
@Override
public int receiveMessage( Message m, DTNHost from ) {
    int       retVal = super.receiveMessage(m, from);

    switch (retVal) {
    case RCV_OK:                                    // the message was received fine
        if (m.getTo().getAddress() != this.getHost().getAddress()) {   // the message is destined for another host; not this
                one
            updateDestinationBids( );
        }

        break;
    case DENIED_OLD:                                // message already received earlier
        if (TRACE_RX) System.out.println( "Denied Old" );
        break;
    case DENIED_TTL:                                // message TTL expired
        if (TRACE_RX) System.out.println( "Denied TTL" );
        break;
    case DENIED_NO_SPACE:                           // no space available for message
        if (TRACE_RX) System.out.println( "Denied No Space" );
        break;
    case TRY_LATER_BUSY:                            // this host is busy receiving or transmitting a message already
        if (TRACE_RX) System.out.println( "Busy, try later");
        break;
    }

    return( retVal );
}


/**
 * Receive message is called when another host sends this host a message. In addition to the what is done in
 */
@Override
public boolean createNewMessage( Message m ) {
    boolean retBool = super.createNewMessage(m);
    m.setTtl( MSG_TTL );                            // set the msg TTL to some predefined period... should be infinity
    updateDestinationBids( );

    return( retBool );
}


/**
 * Method is called just before a transfer is finalized
 * at {@link ActiveRouter#update()}.
 * @param con The connection whose transfer was finalized
 */
@Override
```

```
protected void transferDone(Connection con) {
    Message m = con.getMessage();

    if (m == null) {
        core.Debug.p("Null message for con " + con);
        return;
    }

    this.deleteMessage(m.getId(), false);                // don't leave a copy for the sender
}


/**
 * Updates all bids in this host's destination table.  If a destination doesn't exist, a default destination is created.
 *
 * This will be called in the update() function and updates the movement cost to every destination that exists and that a
       bid
 * is available.
 *
 * @since   March 29, 2014
 * @see     Dst() :: the Dst constructor
 * @see     update()
 * @see     SumHostBids()
 */
private void updateDestinationBids( ) {
    List <Connection> myConnections = getConnections();

    updateRangeStatus();
    updateProximityProbability();
    updateStorageCosts();
    updateTransmissionCosts();

    for (int i=0; i<N_NODES; i++) {
        this.Bm[i] = 0.0;                                // Initialize the parallel bid to 0
        this.minNeighborBid[i] = MAX_BID;
        this.Bs[i] = 0.9 * this.T[i];                    // preferential treatment given to the primary node being the relay
            node.
        this.relayAddress[i] = this.getHost().getAddress();    // set the connected host to be the relay
    }

    for (Connection con : myConnections) {               // loop through all connections; a.k.a. neighboring hosts
        DTNHost nHost = con.getOtherNode( this.getHost() ); // Retrieve the connected host's data
        SelfishRouter_v7 nRouter = (SelfishRouter_v7) nHost.getRouter(); // Use it to retrieve the connected host's router
            information

        // Retrieve a connected host's complete destination list and the list of corresponding calculated bids for each
            destination route.
        // We are actually calculating bids for all neighboring hosts on the host that would normally send the request for
            bids.

        for (int i=0; i<N_NODES; i++) {                  // loop through all possible destinations in the connected hosts'
            tables
            this.Bm[i] += 1.0 / (nRouter.B[i] + this.T[nHost.getAddress()]);

            if (this.T[nHost.getAddress()] < this.minNeighborBid[i]) {
                this.pRelayAddress[i] = nHost.getAddress();
                this.minNeighborBid[i] = this.T[nHost.getAddress()];
            }

            if (nRouter.B[i] + this.T[nHost.getAddress()] < this.Bs[i]) {
                this.Bs[i] = nRouter.B[i] + this.T[nHost.getAddress()];
                this.relayAddress[i] = nHost.getAddress();   // set the connected host to be the relay
            }

            if (!this.isMultipath) this.B[i] = this.Bs[i];
        }
    }

    if (this.isMultipath) {

        for (int i=0; i<N_NODES; i++) {                  // loop through all possible destinations in the connected hosts'
            tables
            this.Bm[i] = 1.0 / this.Bm[i];

            if (this.Bm[i] < this.Bs[i]) {
                this.B[i] = Bm[i];
                this.relayAddress[i] = this.pRelayAddress[i];    // set the connected host to be the relay
            }

            else {
                this.B[i] = this.Bs[i];
            }
        }
    }

    this.lastBidUpdateTime = SimClock.getTime();     // reset the update bid time to be now
    return;
}


private Connection transmitTwoHopMessage( ) {
```

```
    Message             m = null;
    List <Connection>  myConnections = getConnections();

    for (Connection con : myConnections) {      // loop through all connections; a.k.a. neighboring hosts
        DTNHost nHost = con.getOtherNode( this.getHost() ); // Retrieve the connected host's data
        List<Connection> nConnections = nHost.getConnections();

        for (Connection con2 : nConnections) { // loop through all connections; a.k.a. of the neighbors neighboring hosts
            (2-hop)
            DTNHost nnHost = con2.getOtherNode( nHost ); // Retrieve the connected host's data

            m = this.getOldestMessageFromDst( nnHost.getAddress() );  // find the oldest message to destination dChp

            if (m != null) {                        // check to see if there is a message for this destination
                Connection c = getRelayConnection( this.getHost().getAddress(), nHost.getAddress() );

                if (c != null) {
                    if (transmitMessage( m, c )) {
                        return c;
                    }
                }
            }
        }
    }

    return null;
}


/**
 * Transmit the oldest message to the cheapest destination.
  *
 * Find the destination with the cheapest bid
 * Find the oldest message to that destination
 * Check that the connection is up
 *
 *
 * @since April 5, 2014
 */
private void transmitCheapestOldestMessage( ) {
    int             dstAddress;
    Message         oldMsg = null;
    double[]        tmpBid = this.B;

    for (int i=0; i<N_NODES; i++) {         // for every message in the message queue do the following:
        dstAddress = this.findCheapestValidDestination( tmpBid );  // find the cheapest destination in the destination hash
            map

        if (dstAddress != -1) {                         // is it a valid returned destination?
            oldMsg = this.getOldestMessageFromDst( dstAddress );  // find the oldest message to destination dChp

            if (oldMsg != null) {                       // check to see if there is a message for this destination
                // found the cheapest destination and oldest message to that destination, now transfer it
                Connection c = getRelayConnection( this.getHost().getAddress(), this.relayAddress[dstAddress] );

                if (c != null) {
                    if (transmitMessage( oldMsg, c )) break;
                }
            }
        }
    }
}


/**
 * Retrieve the Dst (destination) element and key with the cheapest bid, a valid route and messages to be sent.
 *
 * Note: because of the comparison "dstVal.Cm <= minCm" and because a hash table is used to hold the destinations,
 * there is an inherent priority or preference regarding which destinations will receive messages first if the Cm are
       equal
 *
 * @since April 5, 2014
 */
private int findCheapestValidDestination( double[] tmpBid ) {
    int         min_i = -1;
    double      minCost = Double.POSITIVE_INFINITY;
    double      now = SimClock.getTime();

    for (int i=0; i<N_NODES; i++) {

        if ((this.relayAddress[i] != this.getHost().getAddress()) && (tmpBid[i] < minCost)) {
            // and the bid cost to move the message is cheaper
            minCost = tmpBid[i];                    // found a new minimum movement cost
            min_i = i;                              // save the index as the current offset to the minimum cost value

            if (TRACE_CHEAP_DST) {
                System.out.printf( "Fnd Chp: Now: %-8.2f, Me: %-5d, Dst: %-5d, Rly: %-5d, Bid: %8.2f\n",
                                   now, this.getHost().getAddress(), i, this.relayAddress[i], tmpBid[i] );
            }
        }
    }
```

```
        if (min_i != -1) {                              // we found a minimum value

            if (TRACE_CHEAP_DST) {
                System.out.printf( "Chp Dst: Now: %-8.2f, Me: %-5d, Dst: %-5d, Rly: %-5d, Bid: %8.2f\n",
                                   now, this.getHost().getAddress(), min_i, this.relayAddress[min_i], B[min_i] );
            }

            tmpBid[min_i] = Double.POSITIVE_INFINITY;     // remove this particular destination from contention
        }

        return( min_i );
    }


    /**
     * Find the oldest message targeted for destination d and return it.
     * If one isn't found return NULL;
     *
     * @since April 6, 2014
     * @since April 13, 2014
     */
    private Message getOldestMessageFromDst( int dstAddress ) {
        Message          oldest = null;
        Collection<Message> msgCollection = getMessageCollection(); // load all messages in the queue into this message
            collection

        for (Message m : msgCollection) {                       // traverse all messages in the msg collection

            if (m.getTo().getAddress() == dstAddress) {           // check that the message destination address is the cheapest
                destination

                if (isSending(m.getId())) {                       // check to see if this message is currently being sent
                    continue;                                   // skip the message(s) that router is sending
                }

                if (oldest == null ) {                          // if a message has not been loaded yet
                    oldest = m;                                 // set the message as the oldest by default
                }

                else if (m.getReceiveTime() < oldest.getReceiveTime()) { // find oldest message
//              else if (m.getReceiveTime() > oldest.getReceiveTime()) { // find newest message
                    oldest = m;
                }
            }
        }

        return oldest;
    }


    private boolean transmitMessage( Message m, Connection c ) {

        int retVal = this.startTransfer( m, c );

        if (retVal == RCV_OK) {           // RCV_OK is returned by the receiveMessage( ) function called by the relay node
            return true;
        }

        return false;
    }


    /**
     * @author Michael Stewart
     * Update the Range status of a node. All nodes within a k-hop radius of this node
     * are said to be in range.
     *
     */
    private void updateRangeStatus( ) {

        for (int i=0; i<N_NODES; i++) {         // assume that no packet has arrived in the last Tau period for any
            destination
            this.inRange[i] = false;
        }

        this.inRange[this.getHost().getAddress()] = true;       // a node is always in range with itself
        List <Connection> myConnections = getConnections();

        for (Connection con : myConnections) {       // loop through all connections; a.k.a. neighboring hosts
            DTNHost nHost = con.getOtherNode( this.getHost() ); // Retrieve the connected host's data
            this.inRange[nHost.getAddress()] = true;
        }
    }


    /**
     * Update the connection ratio table
     * @return
     *
     * @log
```

```
 *
 * # Date Time Inits Description
 * 1. 9.3.14   9:45p MFS     It became quite tedious counting all possible paths and then dividing by the
 *                           total number of checks. You would have to divide by the total number of possible
 *                        times a connection could be made but then it isn't very representative of the
 *                        percentage of time 2 nodes are in contact. However, if true / false count is
 *                        implemented such that two nodes are either 2-hop connected or not then it
 *                           is representative of the percentage of time two nodes are connected and extending
 *                           the search to 2 hops just increases the probability that 2 nodes will be in
 *                           contact.
 */
private void updateProximityProbability( ) {
   double now = SimClock.getTime();

   if (now >= lastCcUpdateTime + AUTO_RFB_PERIOD) {

      for (int i=0; i<N_NODES; i++) {

         if (this.inRange[i]) {
            this.connectedCount[i] = this.connectedChecks[i];
         }

         this.connectedChecks[i]++;
         this.theta[i] = this.connectedCount[i] / (double) this.connectedChecks[i];
      }

      lastCcUpdateTime = now;
   }
}


/**
 * Update the storage cost table. The storage cost should only be updated in 2 ways:
 *  1. when a message is transmitted, Cs = (weight) (transmission time - arrival time) + (1 - weight) Cs
 *  2. upon receipt of request for bids, if oldest message is older than SOME_OLD_MESSAGE then Cs = Cs_MAX
 *
 * NOTE: This
 * @return
 */
private void updateStorageCosts( ) {
   double             now = SimClock.getTime();       // get the current time

   for (int i=0; i<N_NODES; i++) {
      Q[i] = 0.0;
      pktCount[i] = 0;
   }

   Collection <Message> msgCollection = getMessageCollection(); // load all messages in the queue into this message
         collection

   for (Message m : msgCollection) {                // traverse all messages in the msg collection
      Q[m.getTo().getAddress()] += now - m.getReceiveTime();
   }
}


/**
 * Update the transmission cost for all destinations in the destination table.
 */

private void updateTransmissionCosts( ) {

   for (int i=0; i<N_NODES; i++) {

      if (this.inRange[i]) {
         this.T[i] = 0.0;
      }

      this.T[i] = ((1 - this.theta[i]) * Q[i]) + this.T[i];
   }
}


/**
 * Find the relay connection
 *
 * @since April 26, 2014
 */

private Connection getRelayConnection( int myAddress, int relayAddress ) {
   List <Connection> myConnections = getConnections();

   for (Connection con : myConnections) {       // loop through all connections; a.k.a. neighboring hosts
      DTNHost nHost = con.getOtherNode( this.getHost() ); // Retrieve the connected host's data

      if ((nHost.getAddress() == relayAddress) &&
         (this.getHost().getAddress() == myAddress)) {

         return con;
      }
   }
```

```
        return null;
    }

    /**
     *
     */
    private void printDstTable( ) {
        double       now = SimClock.getTime();

        for (int i=0; i<N_NODES; i++) {         // loop through all possible destinations in the connected hosts' tables
            System.out.printf( "%-8.2f, Me: %-5d, Rly: %-5d, Dst: %-5d, :: Bid: %-8.3f, Sngl Bid: %-8.3f, Multi Bid: %-8.3f,
                    C: %-8.2f, Theta: %-5.3f, Q: %-8.2f, inRange: %-5b, Cnctd Cnt: %-6d, Cnctd Chks: %-6d\n",
                            now, this.getHost().getAddress(), this.relayAddress[i], i,
                            this.B[i], this.Bs[i], this.Bm[i], this.T[i], this.theta[i], this.Q[i],
                            this.inRange[i], this.connectedCount[i], this.connectedChecks[i] );
        }

        return;
    }


    /**
     * Don't know yet
     */
    @Override
    public SelfishRouter_v7 replicate() {
        return new SelfishRouter_v7( this );
    }

} /** End of Selfish Router class */
```

Appendix B: Simulation Parameters

```
##############################################################
# Selfish Router comparison testing between various routers
# Michael F. Stewart
# January 24, 2015
#
# This scenario includes (SR one-hop version 7)
# It runs through the following buffer sizes (MB): .2, .5, 1, 3, 5, 10, 30
# and uses seeds: (1-53).
# This should be run in batch mode using the following command:
# one -b N MFSthesis_buffersize_scenario.txt where N = 119
##############################################################

## Scenario settings
Scenario.name = %%Group.router%%_%%Group.bufferSize%%_%%MovementModel.rngSeed%%S_%%Group.nrofHosts%%N_%%SelfishRouter.nodeLoopCount%%L
Scenario.simulateConnections = true
# This is in seconds.
Scenario.updateInterval = 0.037
# Scenario runtime
Scenario.endTime = 3600

# "Radio" interface for all nodes
radioInterface.type = SimpleBroadcastInterface

# Transmit speed: 500KBps ~= 5Mbps.  Will base this off
# of typical LTE transfer rates, however this isn't so simple since
# the transfer rate depends on distance between send and receive nodes
radioInterface.transmitSpeed = 10M
# Transmit Range: in meters
radioInterface.transmitRange = 100

# Define 1 node group
Scenario.nrofHostGroups = 1

# Common settings for all groups
Group.movementModel = RandomWaypoint

##################################################################
# THIS WILL CHANGE::: Routing protocol.
# Range from DD, SnW,
##################################################################
Group.router = [SelfishRouter_v7]
##################################################################


##################################################################
# THIS WILL CHANGE::: Buffer sizes.
# This will range from 0.2M, 0.5M, 1M, 3M, 5M, 10M, 20M, 40M
##################################################################
Group.bufferSize = [0.2M; 0.5M; 1M; 3M; 5M; 10M; 30M]
##################################################################

# Basically, the nodes are constantly moving
Group.waitTime = 0, 1
# All nodes have the radio interface
Group.nrofInterfaces = 1
Group.interface1 = radioInterface
# Walking speeds
Group.speed = 0.5, 1.5
# Message TTL of 300 minutes (5 hours)
Group.msgTtl = 300
# Number of nodes in the simulation
Group.nrofHosts = 100
# Use a FIFO queue, required for Prophet Router
Group.sendQueue = 2

# group1 (pedestrians) specific settings
Group1.groupID = p

## Message creation parameters
# How many event generators
Events.nrof = 1
# Class of the first event generator
Events1.class = MessageEventGenerator
# (following settings are specific for the MessageEventGenerator class)
# Creation interval in seconds (one new message every 5 to 10 seconds)
Events1.interval = 1
# Message sizes
Events1.size = 100k
# range of message source/destination addresses
Events1.hosts = 0, 99
# Message ID prefix
Events1.prefix = M

## Movement model settings
```

```
# seed for movement models' pseudo random number generator (default = 0)
################################################################
# THIS WILL CHANGE::: The movement seeds
################################################################
MovementModel.rngSeed = [1;2;3;4;5;6;7;8;9;10;11;12;13;14;15;16;17;18;
                         19;20;21;22;23;24;25;26;27;28;29;30;31;32;33;
                         34;35;36;37;38;39;40;41;42;43;44;45;46;47;48;
                         49;50;51;52;53]
################################################################

# World's size for Movement Models without implicit size (width, height; meters)
MovementModel.worldSize = 1000, 1000
# How long time to move hosts in the world before real simulation
MovementModel.warmup = 100

## Reports - all report names have to be valid report classes
# how many reports to load
Report.nrofReports = 6
# length of the warm up period (simulated seconds)
Report.warmup = 0
# default directory of reports (can be overridden per Report with output setting)
Report.reportDir = reports/
# Default settings for reports
MessageLocationReport.granularity = 60
MessageLocationReport.messages = M
# Report classes to load
Report.report1 = MessageStatsReport
Report.report2 = CreatedMessagesReport
Report.report3 = DeliveredMessagesReport
Report.report4 = ContactTimesReport
Report.report5 = AdjacencyGraphvizReport
Report.report6 = MessageLocationReport


## Default settings for some routers settings
MaxPropRouterWithEstimation.timeScale = 10
ProphetRouterWithEstimation.timeScale = 10
ProphetRouter.secondsInTimeUnit = 10
ProphetV2Router.secondsInTimeUnit = 10
SprayAndWaitRouter.nrofCopies = 6
SprayAndWaitRouter.binaryMode = true
SelfishRouter.multiPathMode = false
SelfishRouter.nodeLoopCount = 5

## Optimization settings -- these affect the speed of the simulation
## see World class for details.
Optimization.cellSizeMult = 5
Optimization.randomizeUpdateOrder = true

## GUI settings

# GUI underlay image settings
#GUI.UnderlayImage.fileName = data/helsinki_underlay.png
# Image offset in pixels (x, y)
GUI.UnderlayImage.offset = 64, 20
# Scaling factor for the image
GUI.UnderlayImage.scale = 4.75
# Image rotation (radians)
GUI.UnderlayImage.rotate = -0.015

# how many events to show in the log panel (default = 30)
GUI.EventLogPanel.nrofEvents = 100
# Regular Expression log filter (see Pattern-class from the Java API for RE-matching details)
#GUI.EventLogPanel.REfilter = .*p[1-9]<->p[1-9]$
```

Vita

Michael Stewart was born in New Orleans, Louisiana on January 27, 1973 to Francis and Judith Stewart. After graduating high school in 1991 he worked before returning to university in earnest in 1995. He received a B.S. in Physics from Louisiana State University in May 1999 and continued his research as a fulltime employee with the Department of Physics and Astronomy. He expects to receive his M.S. in Systems Science from Louisiana State University in December 2015.